Reducing memory consumption of ProVerif with hash consing techniques

Margot Catinaud, under supervision of Vincent Cheval at PROSECCO / Inria Paris

March - August 2022

The general context

Cryptographic protocols aim at securing communications. They are used in various applications: secure message establishment, electronic voting, mobile communications, etc. However, there are three main difficulties to design such a secure protocol. First one is about cryptography, we need secure enough cryptographic primitives, *i.e.* reductible to sufficiently hard problems such as the *discrete logarithm problem* or the *number factorization problem*. Second one is about hardware, there may exist security flaws in the implementation of protocols (such as *Heartbleed* in OpenSSL) or in design of hardware (such as *SPECTRE* or *Meltdown* attacks on Intel processors). Finally, the third one is about designing a protocol itself. Indeed, there exists attacks such as *man-in-the-middle* attacks which can by-pass the security of cryptographic primitives or leak cryptographic keys. For example, older versions of the TLS (*Transport Layer Security*) protocol, which is used to secure internet browsing (the 'S' of `HTTPS`) have the three types of flaws listed above. Therefore, it has become common practice to analyse the security of a protocol using formal methods and in particular automatic tools. In this paper, we will focus on the third type of vulnerability, namely those introduced at the design stage of the protocol itself. Several tools have been proposed for automatized security analysis of protocols, ProVerif is one of them [3].

This tool is often preferred for large and complex protocols such as TLS [2] or Signal [12]. In order to check whether a property of security is verified or not, we have to give a model for the attacker. In the litterature, there exists three main models for the attacker. The first one is the model of Dolev-Yao, which is only symbolic. In this model, we give to the attacker the ability to compute any function they want and generate any value; this is the model used by ProVerif. The second model corresponds to a computational model where the attacker is a probabilistic polynomial-time Turing machine. The tool CryptoVerif, developped by Bruno Blanchet [4], use this model. Finally, the third one is the model of Bana-Common which is a symbolic approach in the computational model. In this model, we talk about a computationally complete symbolic attacker. The tool Squirrel, developped by Adrien Koutsos [1], uses this last model.

Problem studied

Actually, verifying real-life protocols costs a lot of machine resources, especially memory. As an order of magnitude, the TLS protocol with Encrypted ClientHello (ECH) extension can use between 10 and 100 GB of memory depending on the security properties we want to prove. Moreover, proving security properties is a difficult problem that is not specific to ProVerif. In fact, it is an undecidable problem in general but under some conditions it is a CO-NEXP problem. It therefore appears necessary to optimise the automatic verification tools both in terms of time and memory in order to be able to go further in the verification of protocol

prototypes. In the case of ProVerif, given a protocol and a security property, it may either prove that the property is satisfied or exhibit an attack. It may also return "cannot be proved" meaning that it can not reach a conclusion. Recent improvements to ProVerif, such as the addition of lemmas and axioms [9], have resulted in a significant speed-up of resolutions (by a factor of 30 to 40). This memory problem is crucial because of computers capacities limits. This is why the objective of my internship is to propose ProVerif improvements on memory consumption using hash consing techniques.

Proposed contributions

We have added a new internal library to ProVerif which allows us to adapt the classic techniques of *hash consing* to the internal representation of terms, thus maximising memory sharing. In particular, with this new library, two semantically equal messages will be guaranteed to be physically equal (they will point to the same address in memory). Once this representation was added, we had to adapt the classical operations of ProVerif on terms such as matching or unification. The internal algorithm of ProVerif being mainly based on the saturation of Horn clauses, a maximal sharing of the memory within each clause but also within the whole set of Horn clauses generated by ProVerif allows us to considerably reduce the memory consumption. While maximizing memory sharing within a clause is fairly straightforward, for Horn clauses sets, the problem becomes much more difficult due to the presence of variables that may have to be considered as distinct in two different clause sets.

The results of implementations are available on the following repository gitlab: *https://gitlab.inria.fr/bblanche/proverif* on the branch *hashconsing*.

Arguments supporting its validity

A formal framework has been proposed to formalise algorithms that perform operations on *hash-consed* terms. We were able to test the validity of this framework by giving a proof of correctness for the new version of the algorithm performing syntactic unification. Furthermore, it has been shown that the new data structure for terms and adapted algorithms saves between two and ten times more memory. These differences in gain depend very much on the protocol as well as the type of security properties one is trying to prove. If *hash consing* techniques are costly in time, the improvement of unification and matching algorithms allows us to mitigate this cost. For example, in the case of TLS + ECH, for the same set of tests to resolve certain security properties, while the old version of ProVerif required between 1 and about 150 GB of memory, the new version requires only between 1 and 6 GB. Actually, the very design of this protocol implies that there is a lot of redundancy in the messages, this is why this protocol naturally invites memory sharing, which is indeed shown by the performance tests.

Summary and future work

During my internship, we proposed a new structure for the terms *hash consed* which gives us an equivalence between physical equality and syntactic equality. Thus, the equality on the terms is now in constant time. In addition, we adapted some algorithms that could be improved with the help of the new term structure. Moreover, we have proposed a prototype for sharing variables in a set of Horn clauses. This being said, some improvements can still be made, such as adapting more algorithms in order to make less conversions between the terms and the classical ones. Moreover, the proposal of sharing variables within a set of Horn clauses is currently time efficient but not optimal in term of memory space.

I Presentation of ProVerif

A cryptographic protocol can be seen as a recipe for what messages to exchange in order to establish a secure communication between a client and a server. These messages are exchanged over a network (such as the Internet) that is assumed to be controlled by an attacker. As an example, in the case of the *man-in-the-middle* attack where the attacker is an intermediary on the network between the client and the server. In the Dolev-Yao model, the messages exchanged by the protagonists of a protocol correspond to terms in which the cryptographic primitives are seen as unbreakable black boxes.

Let us present an example of a toy protocol, the Needham-Shroeder protocol which consists basically in a handshake.

$$\begin{array}{lcl} A \longrightarrow B & : & \texttt{enc}((A, n_A), \texttt{pk}(B)) \\ B \longrightarrow A & : & \texttt{enc}((n_A, n_B), \texttt{pk}(A)) \\ A \longrightarrow B & : & \texttt{enc}(n_B, \texttt{pk}(B)) \end{array}$$

Figure 1: The Needham-Shroeder protocol

Our toy protocol uses an public-key encryption/decryption scheme which corresponds to an idealised version of cryptographic primitives. In the symbolic model, these primitives are modelled by inference rules. As an example, if a cypher message $m = \texttt{enc}(x, \texttt{pk}(A))$ is encrypted by a public-key $\texttt{pk}(A)$, we can deduce the original message x from m as soon as we know the private-key $\texttt{sk}(A)$. This corresponds to the following inference rule:

$$\frac{\texttt{enc}(x, \texttt{pk}(A)) \qquad \texttt{sk}(A)}{x}\ \texttt{dec}$$

One security property that comes naturally from the definition of the Needham-Shroeder protocol is the following: *Is the attacker able to deduce n_B from the messages they saw on the network?* This is an example of *secrecy* property (we will talk later about different type of security property which can be proved by ProVerif). In the Dolev-Yao model, we assume that the protocol is executed in presence of an attacker, model by predicate $\texttt{att}(\cdot)$, that can intercept all messages, compute new messages from the messages it has received and send any message it can build.

Here is a diagram that summarises the core of the algorithm that ProVerif uses to verify security properties on a given protocol.

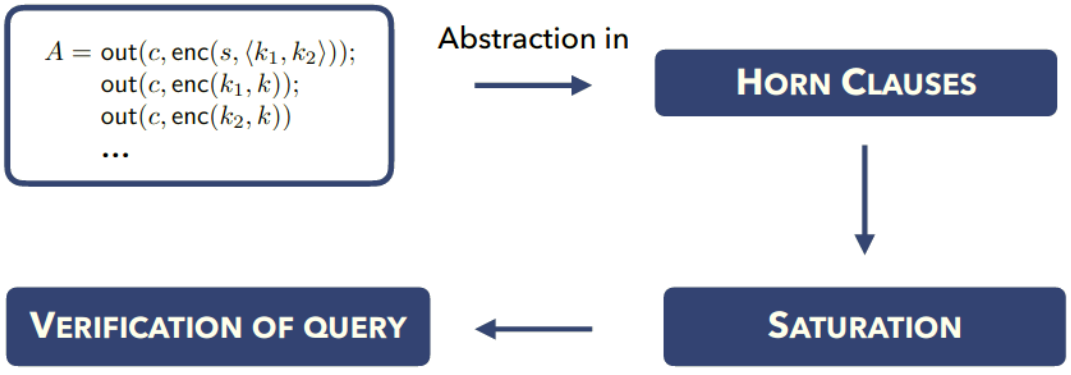

Figure 2: ProVerif's algorithm in a nutshell [10]

To model protocols, ProVerif comes with its own core language, an extension of pi-calculus, whose syntax is specified in annex A. With this extension, it is the terms that are communicated over the network for which standard operational semantics have been added to model what is transiting and the inputs are terms constructed by the attacker from their initial knowledge and previous outputs. Once the protocol and queries for security properties has been written in the ProVerif language, it is abstracted into Horn clauses. The syntax for these clauses is as follows:

$M, N ::=$	terms
x	variable
$a[M_1, \ldots, M_n]$	name
$f(M_1, \ldots, M_n)$	function application
$F ::= p(M_1, \ldots, M_n)$	fact
$R ::= F_1 \wedge \ldots \wedge F_n \Rightarrow F$	Horn clause

Figure 3: Syntax of Horn clauses

In our current protocol example, its transformation in Horn clauses includes the following clauses:

Computation abilities for the attacker

encryption	$\texttt{att}(x) \wedge \texttt{att}(k) \Rightarrow \texttt{att}(\texttt{enc}(x, k))$
decryption	$\texttt{att}(\texttt{enc}(x, k)) \wedge \texttt{att}(k) \Rightarrow \texttt{att}(x)$
pairing	$\texttt{att}(x) \wedge \texttt{att}(y) \Rightarrow \texttt{att}((x, y))$
proj-left	$\texttt{att}((x, y)) \Rightarrow \texttt{att}(x)$
proj-right	$\texttt{att}((x, y)) \Rightarrow \texttt{att}(y)$
public key	$\texttt{att}(id) \Rightarrow \texttt{att}(\texttt{pk}(id))$

The protocol

initial knowledge	$\Rightarrow \texttt{att}(\texttt{pk}(A[])), \Rightarrow \texttt{att}(\texttt{pk}(B[]))$
first message	$\Rightarrow \texttt{att}(\texttt{enc}((A[], n_A[]), \texttt{pk}(B[])))$
second message	$\texttt{att}(\texttt{enc}((x, y), \texttt{pk}(B[]))) \Rightarrow \texttt{att}(\texttt{enc}((y, n_B[\texttt{enc}((x, y), \texttt{pk}(B[]))]), x))$
third message	$\texttt{att}(\texttt{enc}((n_A[], y), \texttt{pk}(A[]))) \Rightarrow \texttt{att}(\texttt{enc}(y, \texttt{pk}(B[])))$

The query(ies) $\texttt{att}(n_B)$.

Figure 4: Result of the transformation in Horn clauses for the Needham-Shroeder protocol

The next step is called *saturation* and is based on the following resolution inference rule:

$$\frac{H \wedge F \Rightarrow C \qquad H' \Rightarrow C' \qquad \sigma = mgu(F, C')}{H\sigma \wedge H'\sigma \Rightarrow C\sigma} \text{ (Res)}$$

where *mgu* is for *most-general unifier*, its definition will be given in subsection IV.1.

Let us focus more on the saturation process that is applied until a fixpoint is found. This core algorithm consists of several steps.

- **Step 1: Resolution** Basically, this step consists in application of the rule *(Res)*. However for a clause where the attacker can apply a function h, $\texttt{att}(x) \Rightarrow \texttt{att}(h(x))$, resolution can

be applied as much as we want, producing this way the facts $\texttt{att}(h^n(x))$ for any integer $n \in \mathbb{N}$. Hence, we need to guide resolution step by a selection function $\texttt{sel}$ which is a function from clauses to sets of facts, such that $\texttt{sel}(H \Rightarrow C) \subseteq H$. If $F \in \texttt{sel}(R)$, we say that F is selected in R. If $\texttt{sel}(R) = \varnothing$, we say that no hypothesis is selected in R, or that the conclusion of R is selected. Actually, the selection function used in ProVerif is the following:

$$\texttt{sel}(H \Rightarrow C) = \begin{cases} \varnothing & \text{if for all fact } F \in H, \text{ there exists a variable } x \text{ such that} \\ & \quad F = \texttt{att}(x) \\ \{F_0\} & \text{where } F_0 \in H \text{ and for all } x \text{ variable, } F_0 \neq \texttt{att}(x) \end{cases}$$

Hence the resolution rule becomes

$$\frac{H \wedge F \Rightarrow C \quad H' \Rightarrow C' \quad \sigma = mgu(F, C') \quad F \in \texttt{sel}(H \wedge F \Rightarrow C) \quad \texttt{sel}(H' \Rightarrow C') = \varnothing}{H\sigma \wedge H'\sigma \Rightarrow C\sigma}$$

- **Step 2: Simplification** If we apply resolution alone, it will not terminate because the current set of clauses will always increase. However, care must be taken to ensure that the simplifications to the clause are sound. For instance, we can remove *tautologies*, *i.e.* clauses of the form $H \wedge F \Rightarrow F$. Another example is simplifying facts corresponding to knowledge of the attacker under some conditions, *i.e.* simplify clauses of the form $H \wedge \texttt{att}(x) \Rightarrow C$ into $H \Rightarrow C$ when x appears nowhere else in the clause.
- **Step 3: Elimination** Finally, it may be that a clause *subsumes* (*i.e.* is more general than) the clause R' we want to add to the current set of Horn clauses. If this is the case, there is no need to add R', we can eliminate it. Otherwise, we add R' to the set and eliminate the clauses *subsumed* by this clause.

To summarise the saturation procedure, a diagram from a presentation on the verification of cryptographic protocols [10] is shown in figure 5.

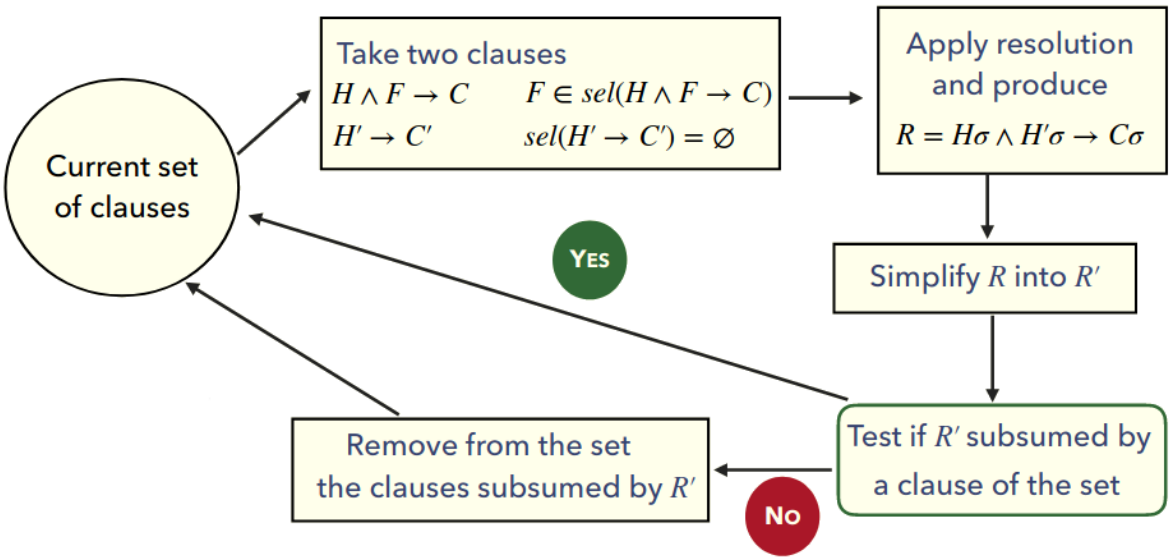

Figure 5: Summary of the saturation procedure

Finally, we can verify whether the set of clauses corresponding to the query(ies) are derivable from the saturated set of clauses. In particular, as an example, we say secrecy of some message s is preserved if the fact $\texttt{att}(s)$ is not logically deductible from the set of Horn clauses. If a derivation is found, there may exists an attack on the protocol and otherwise, the security property holds!

Actually, ProVerif can verify various security properties, by using an adequate translation into Horn clauses:

- *secrecy* properties: the adversary cannot compute certain values.
- *authentication* properties: if some participant Alice thinks she talks to Bob, then she really talks to Bob. Authentication is formalized by correspondance properties of the form: if some event has been executed, then some other event has been executed. Events can represent that Alice concluded the protocol apparently with Bob, or that Bob started the protocol, apparently with Alice.
- *equivalence* properties: the adversary cannot distinguish an implementation of a protocol from its specification, hence its ensures privacy properties. Equivalence properties are a powerful notion to specify security properties (such as anonymity and privacy), but they are also difficult to verify. ProVerif can verify only a strong notion of equivalence, named *diff-equivalence*, between protocols that have the same structure but differ only by the messages they exchange.

II Hashconsing

As we saw in previous section, Horn clauses are defined by terms. Let $\mathcal{X}$ be an infinite set of variables. Let $\mathcal{F} = \{f_i/n_i \mid i \in I\}$ be a finite signature where $I \subset \mathbb{N}$ is a finite set and for $i \in I$, f_i/n_i represents a function symbol f_i with its arity n_i. We define terms as variables or application of a function symbol on terms, denoted $\mathcal{T}(\mathcal{X}, \mathcal{F})$. Actually, terms have a structure of trees where leaf are either variables or function symbols of arity 0 and nodes are application of a function symbol. For instance, if $b, x, c, k \in \mathcal{X}$ and $\mathcal{F} = \{\texttt{enc}/2, \langle\cdot,\cdot\rangle/2, \oplus/2\}$ then the term $t = \texttt{enc}(b \oplus \langle x, c\rangle, k)$ is represented as following:

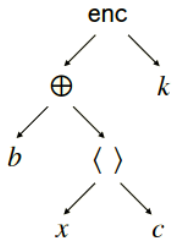

Figure 6: Example of representation of term t in tree

In OCaml, the definition of generic terms in $\mathcal{T}(\mathcal{X}, \mathcal{F})$ will be:

```
type term =
    | Var of variable
    | FunApp of fun_symb * term list
```

However, as OCaml does not natively support memory sharing, we may have a term with two syntactically (*i.e.* structurally equal) but not physically equal subterms. For a protocol

such as TLS, its design implies the existence of many syntactically equal subterms. This is why the current version of ProVerif uses between 10 and 100 GB of memory for this protocol.

To perform a compression of terms in a Horn clause, we will use *hash consing* techniques as described in [11], but we need to adapt them to the specifications of ProVerif. In fact, generic variables in ProVerif are associated to a given type and are identified by an unique id. Hence the structure of a term can not be anymore a syntactic tree but an acyclic oriented graph.

Definition II.1 (Term-graph) *A* term-graph G *is an acyclic oriented graph such that:*

- *nodes of* G *are labeled by either a variable from* $\mathcal{X}$ *or a function symbol from* $\mathcal{F}$
- *a node labeled by* $f/n \in \mathcal{F}$ *has* n *children*
- *a node labeled by* $x \in \mathcal{X}$ *is a leaf*

Given a node η *from* G*, we will denote by* $\lambda_G(\eta)$ *its label and by* $\gamma^i_G(\eta)$ *the* i*-th child of* η*. When clear from context, we will omit* G *and write* $\lambda(\eta)$ *and* $\gamma^i(\eta)$*.*

For instance, the terms $f(x, g(y, z))$ and $f(g(y, z), g(a, a))$ (with $a/0, f/2, g/2 \in \mathcal{F}$ and $x, y, z \in \mathcal{X}$) are represented by the following term-graph:

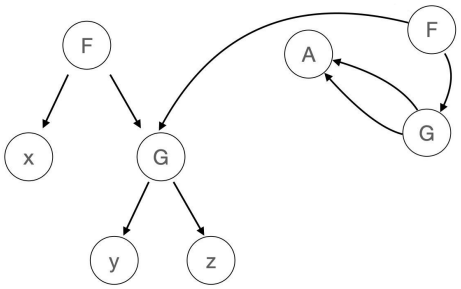

Figure 7: Example of term-graph

Moreover, we do not want to use more storage for any algorithm or function on *hash consed* terms. Thus, we will define the notion of links associated to term-graphs which will allow to store results in the graph itself. We implement those links by a special type defined by:

```
type hcterm_link =
    | HCNoLink
    | HCVisited of bool
    | HCTerm of hcterm
    | HCResult of hcterm
    | HCResultInt of int
    | HCResultIntInt of int * int
    | HCResultTermInt of hcterm * int
    | HCBasicTerm of term
```

Hence, in OCaml, a *hash consed* term will be modeling as a record:

```
type hcterm = {
    hc_tag: int;
```

```
    hc_desc: hcterm_desc;
    mutable hc_link: hcterm_link;
}
```

Regarding the ability for the *garbage collector* of OCaml to reclaim the *hash consed* terms that are not referenced anymore (from anywhere else than the *hash consed* table), the solution is to use weak pointers. That's why we will use the functor

```
Weak.Make (H: Hashtbl.HashedType)
```

Thus, we implement hash table for *hash consed* terms in this way:

```
module HashConsing = struct
    type t = hcterm

    let equal hct1 hct2 = match hct1.hc_desc,hct2.hc_desc with
        | HCVar hcv, HCVar hcv' -> hcv == hcv'
        | HCFunApp(f1,args1), HCFunApp(f2,args2) ->
            f1 == f2 && List.for_all2 (==) args1 args2
        | _ -> false

    let hash hct = match hct.hc_desc with
        | HCVar hcv -> hcv.hc_vname.idx
        | HCFunApp(f,args) ->
            List.fold_left (fun acc hct -> acc * 65599 + hct.hc_tag)
                           (f.f_record) args
end

module HashConsingTbl = Weak.Make (HashConsing)
```

In fact, this implementation was heavily inspired from the paper on *hash consing* techniques [11] but we have proposed our own functions `equal` and `hash`. These new functions are more efficient because they assume that the subterms have already been *hash consed*, hence they only look at the current node without following its childrens. The field `hc_tag` in `hcterm` type is an integer that uniquely identifies the term; this is why it is used to calculate the *hash* of a `hcterm`.

Thanks to implementation of the `Weak` library, we have the following theorem:

Theorem II.1 *For* hash consed *terms, we have an equivalence between physical equality (denoted by `( == )`) and syntactic equality (denoted by `( = )`).*

In particular, all the algorithms that we are going to implement on *hash consed* term will process each *syntactically* different subterms at most once. This is why we have added all these constructors to the type of link `hcterm_link`.

Representation in Horn clauses is not unique, a clause $R = H \Rightarrow C$ is invariant by renaming variables, obviously ensuring if two variables are distinct before renaming, then these two variables remains distinct after. Because of this invariant, we want to share variables as much as possible inside a given set of Horn clauses. Remember the saturation process. The first step is to choose two clauses R and R' such that the first one is such that $R = H \wedge F \Rightarrow C$ with $F \in \texttt{sel}(H \wedge F \Rightarrow F)$ and the second one is such that $R' = H' \Rightarrow C'$ with $\texttt{sel}(H' \Rightarrow C') = \varnothing$. A clause such as R is called *unsolved* and a clause such as R' is called *solved*. Consequently we have at least two databases, *i.e.* set of clauses, one for unsolved and another one for solved.

Procedure to convert a generic term to a *hash consed* term. In order to convert terms, we have to use a *hash* table. Indeed, this table have its own accumulator which is an integer identifying in a unique way each *hash consed* term. In order to build an element of type `hcterm`, we start to build an element with a temporary `hc_tag` set to the value of the accumulator. As the function `hash` does not use this field, we can lookup in the *hash* table associated to the database $\mathbb{D}$ if the element is already present. If it is, we return the corresponding element of the table. If not, the temporary tag is a good one, so we can increment the database accumulator value by 1. We denote by `build_node` the *smart constructor* of type

```
build_node : fun_symb -> hcterm list -> hcterm
```

performing *hash consing* for nodes corresponding to function application. On the other hand, we note by `fresh_hcvariable` the *smart constructor* of type

```
fresh_hcvariable : bool -> renamable_id -> typet -> hcterm
```

performing *hash consing* for nodes corresponding to variables.

Procedure to add a clause R to a given database $\mathbb{D}$. The database $\mathbb{D}$ of clauses is also accompagnied by a set of variables $\mathcal{X}(\mathbb{D})$ used in clauses already present and which is maintained through the lifetime of $\mathbb{D}$. The main idea is to copy the clause R from the database where it belongs to the new database while taking care to rename the variables in order to maximise sharing. To do so, first we need to duplicate the set of variables $\mathcal{X}(\mathbb{D})$ into a second set $\mathcal{X}_0$. Then, the first time we see a given variable x in R we have two cases. If there is no more variable in the set $\mathcal{X}_0$, *i.e.* $\mathcal{X}_0 = \varnothing$, we must create a new variable $\tilde{x}$ which we will immediatly add to the set $\mathcal{X}(\mathcal{S})$. Otherwise, we choose a variable $\tilde{x} \in \mathcal{X}_0$ which we remove from this set immediatly. Finally, we replace x by $\tilde{x}$ in R. This is how the new version of the R clause is built as we go along, also making good use of the `build_node` function. In fact, during this procedure, all intermediary results are stored thanks to the link [`HCResult`] which makes it possible to optimise the procedure in time. When all variables R are renamed, we add this new clause to the database $\mathbb{D}$ and we delete all links corresponding to intermediary results (we will explain later how this cleaning works).

Indeed, this procedure does not allow to completely optimize the number of subterms of a clause database. The idea of the following problem is then to give a formalization to a compression algorithm that would allow to compress the number of different subterms within the same database.

Problem *Let $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{X}, \mathcal{F})$ be n terms. Find the set of bijective renaming functions $\{\rho_i\}_{i=1}^n$ that minimises the quantity*

$$\texttt{Card}\left(\bigcup_{i=1}^{n} \texttt{Subterms}(t_i\rho_i)\right).$$

IV Operations on *hash consed* terms

IV.1 Formal framework

Now that we have a suitable structure for memory sharing, it is time to adapt the main ProVerif's algorithms in order to take full advantage of our new term representation. Hence, in this section we will start by give a formal framework to ensure correctness of the adapted algorithms. As shown by diagram given in figure 5, some algorithms are crucial for saturation: for example unification both syntactic and modulo an equational theory.

As we have chosen to store results of our functions with the mutable type `hcterm_link` directly in our structure of `hcterm`, we need to formalise *side effects* of our functions. Let us start by giving a formal definition for our links, with the definition of linking functions. Notice that as we will only be dealing with unification, the following definition for the linking functions contains only those links that are useful for unification, namely the constructors [`HCNoLink`], [`HCTerm`] and [`HCVisited`].

Definition IV.1 (Linking function) *Given a term-graph G, we say δ* is a partial linking function for G *when the domain of δ is a subset of nodes of G and for all $\eta \in \mathtt{dom}(\delta)$,*

$$\delta(\eta) \in \left\{\Lambda^T(\eta') \mid \eta' \in G\right\} \cup \left\{\Lambda^V(b) \mid b \in \{\top, \bot\}\right\}$$

where

- $\Lambda^V(\cdot)$ *is used to mark a visited node during a* depth-first search *algorithm*
- $\Lambda^T(\cdot)$ *denote a link toward another node.*

We denote by $\Delta(G)$ the set of linking functions for G.

For a function `func` with n arguments $\left(\eta_i\right)_{i=1}^n$ over nodes of a term-graph G and with a linking function $\delta \in \Delta(G)$, we denote by $\mathtt{func}\left[\delta\right]\left(\eta_1, \ldots, \eta_n\right)$ the application of `func` with context δ. We denote by $\mathtt{res}\left[\delta\right]$ the result of a function. Therefore, if we apply a function `func` on n nodes $(\eta_i)_{i=1}^n$ in the original context corresponding to the linking function δ and whose call result is `res` in a new context δ' modelling the edge effects having taken place at the time of the call to this function `func`, we will write:

$$\mathtt{func}\left[\delta\right]\left(\eta_1, \ldots, \eta_n\right) = \mathtt{res}\left[\delta'\right].$$

In reality, the resulting linking function δ' is not constructed in any way, it "derives" from the original linking function δ by keeping a certain order.

Definition IV.2 (Partial order on linking functions) *We define a partial ordered relation on $\Delta(G)$ by*

$$\delta \preccurlyeq \delta' \quad \Longleftrightarrow \quad \begin{cases} \mathtt{dom}(\delta) \subseteq \mathtt{dom}(\delta') \\ \forall\, \eta \in \mathtt{dom}(\delta), \left[\exists\, \eta' \in G, \delta(\eta) = \Lambda^T(\eta')\right] \implies \delta'(\eta) = \delta(\eta). \end{cases} \tag{1}$$

Let η be a node of a term-graph G. Let us define some notations. First, we denote by $\mathtt{term}_G(\eta)$ the term in $\mathcal{T}(\mathcal{X}, \mathcal{F})$ corresponding to subgraph of G with η as root without following any linking function: if $\lambda_G(\eta) \in \mathcal{X}$ then $\mathtt{term}_G(\eta) = \lambda_G(\eta)$ and otherwise, *i.e.* $\lambda_G(\eta) = f/n \in \mathcal{F}$, $\mathtt{term}_G(\eta) = f(\mathtt{term}_G(\gamma_G^1(\eta)), \ldots, \mathtt{term}_G(\gamma_G^n(\eta)))$.

Then, given a linking function δ, we want to introduce notations in order to take it into account. We denote by $\Gamma_G^\delta(\eta)$, for $\eta \in G$, the set of nodes connected by the linking function δ: if $\eta \in \mathtt{dom}(\delta)$ and there exists $\eta' \in G$ such that $\delta(\eta) = \Lambda^T(\eta')$ then $\Gamma_G^\delta(\eta) = \{\eta\} \cup \Gamma_G^\delta(\eta')$ and otherwise, $\Gamma_G^\delta(\eta) = \{\eta\}$. Given η, it may be linked to a node which is itself linked etc. We therefore denote by $\mathtt{find}_G^\delta(\eta)$ the node in G obtained by following links given by function δ: if $\eta \in \mathtt{dom}(\delta)$ and there exists $\eta' \in G$ such that $\delta(\eta) = \Lambda^T(\eta')$ then $\mathtt{find}_G^\delta(\eta) = \eta'$ and otherwise, $\mathtt{find}_G^\delta(\eta) = \eta$.

Finally, the term corresponding to subgraph with root η in G where all the links are followed is denoted by $\mathtt{follow}_G^\delta(\eta)$ and it is defined by: if $\mathtt{find}_G^\delta(\eta) = \eta'$ with $\lambda_G(\eta') \in \mathcal{X}$ then $\mathtt{follow}_G^\delta(\eta) = \eta'$ otherwise, *i.e.* if $\mathtt{find}_G^\delta(\eta) = \eta'$ with $\lambda_G(\eta') = f/n \in \mathcal{F}$, $\mathtt{follow}_G^\delta(\eta) = f(\mathtt{follow}_G^\delta(\gamma_G^1(\eta')), \ldots, \mathtt{follow}_G^\delta(\gamma_G^n(\eta')))$.

However, it can be seen that the quantities $\mathtt{find}_G^\delta(\eta)$ and $\mathtt{follow}_G^\delta(\eta)$ may not be well-defined. We therefore need to establish a notion of well-foundation for the linking functions.

Definition IV.3 (Well-foundation) *Let δ be a linking function for G. We say* δ is well-founded *when δ generates no cycle,* i.e. *when $\texttt{find}^{\delta}_{G}(\eta)$ is well-defined for all node $\eta \in G$ and δ link no node to $\Lambda^V(\bot)$ or $\Lambda^V(\top)$,* i.e. *for all $\eta \in \texttt{dom}(\delta)$, $\delta(\eta) \notin \{\Lambda^V(\top), \Lambda^V(\bot)\}$.*

Unfortunately, this definition is not sufficient for the quantity $\texttt{follow}^{\delta}_{G}(\eta)$ to be well-defined. Indeed, it is possible that a cycle following both the links given by the linking function δ and the edges of the term-graph G appears. Let us start by defining a relation $\rightarrow_\delta$ on the nodes that expresses the fact that a first node is connected to another one either by following a link or by following an edge. For all nodes $\eta, \eta' \in G$ such that $\eta \rightarrow_\delta \eta'$ then either $\delta(\eta) = \Lambda^T(\eta')$ or $\eta \notin \texttt{dom}(\delta)$, $\lambda_G(\eta) = f/n \in \mathcal{F}$ and there exists $i \in [\![1\ ;\ n]\!]$ such that $\eta' = \gamma^i_G(\eta)$. We write $\rightarrow^*_\delta$ for the reflexive and transitive closure of $\rightarrow_\delta$. Considering two nodes η and η' of G such that $\eta \rightarrow^*_\delta \eta'$ (respectively $\eta \rightarrow^+_\delta \eta'$), we say η' is *δ-reachable* (resp. *strictly δ-reachable*) from η.

Definition IV.4 (δ-cycle) *Let η be a node of a term-graph G and δ a linking function of G. We say there is a* δ-cycle in G δ-reachable from η *when there exists a node $\eta' \in G$ such that η' is δ-reachable from η and η' is strictly δ-reachable from η',* i.e. $\eta \rightarrow^*_\delta \eta' \rightarrow^+_\delta \eta'$. *In addition, we say G* is δ-cycle free *if for all $\eta \in G$ there is no δ-cycle in G δ-reachable from η.*

We notice that if a term-graph G is δ-cycle free, then G does not contains cycles of links. Thus, δ is well-founded. Moreover, we have the following lemma

Lemma IV.1 *Let $\eta \in G$ be a node of a term-graph G and δ be a well-founded linking function. Suppose there is no δ-cycle in G δ-reachable from η. Then the quantity $\texttt{follow}^{\delta}_{G}(\eta)$ is well-defined.*

IV.2 Syntactic unification

Remember the resolution step during saturation of a set of Horn clause. This step uses the following rule:

$$\frac{H \wedge F \Rightarrow C \qquad H' \Rightarrow C' \qquad \sigma = mgu(F, C') \qquad F \in \texttt{sel}(H \wedge F \Rightarrow C) \qquad \texttt{sel}(H' \Rightarrow C') = \varnothing}{H\sigma \wedge H'\sigma \Rightarrow C\sigma}$$

In fact, this rule uses syntactic unification, the first algorithm discussed in this very subsection. A substitution is a function σ mapping variables to terms $\sigma : \mathcal{X} \longrightarrow \mathcal{T}(\mathcal{X}, \mathcal{F})$. An application of a substitution over a term is naturally defined by induction over the term in which variables are replaced by terms. We say two terms $u, v \in \mathcal{T}(\mathcal{X}, \mathcal{F})$ are *(syntactically) unifiable* if and only if there exists a substitution σ such that $u\sigma = v\sigma$. An unifier σ is called the *most-general unifier* of terms $u, v \in \mathcal{T}(\mathcal{X}, \mathcal{F})$ if for any other unifier θ of these two terms then there exists a substitution θ' such that $\theta = \sigma \circ \theta'$.

Within a term-graph G, a substitution is modelled by the links [HCTerm]. Given a linking function δ, we then define the notion of *substitution induced by δ* as follows:

$$\texttt{subst}(\delta) = \Big[\lambda_G(\eta) \mapsto \texttt{follow}^{\delta}_{G}(\eta) \mid \eta \in \texttt{dom}(\delta) \quad \wedge \quad \lambda_G(\eta) \in \mathcal{X} \quad \wedge \quad \exists\, \eta' \in G, \delta(\eta) = \Lambda^T(\eta')\Big] \tag{2}$$

The following lemma then expresses that this notion is indeed what was expected.

Lemma IV.2 *Let η be a node of a term-graph G and δ a well-founded linking function. Suppose G without δ-cycle. Then*

$$\texttt{follow}^{\delta}_{G}(\eta) = \texttt{term}_G(\eta)\texttt{subst}(\delta).$$

To unify two terms $u, v \in \mathcal{T}(\mathcal{X}, \mathcal{F})$, represented in a term-graph G, we will use an improved version of syntactic unification. Indeed, when we wish to unify a variable with a function application, it is necessary to ensure that we do not introduce a cycle, this is the *occur-check* procedure. With this procedure, the syntactic unification algorithm can be exponential in the number of nodes of the term-graph G in the *worst case* scenario. To reduce this complexity to a quadratic algorithm, the syntactic unification algorithm can be divided into 2 main steps (as specified in slides [13] about the speeding up the unification algorithm):

1. Unify the terms without worrying about whether or not a cycle is introduced, this is the `cyclic_unify` procedure.

2. Check if a cycle has been introduced, this is the `no_cycle` procedure.

IV.2.1 Step 1: Unification procedure that can introduce cycles

We suppose we have a function `add_new_link` such that if δ is a linking function, $\eta, \eta' \notin \texttt{dom}(\delta)$ two nodes of G then $\texttt{add\_new\_link}\left[\delta\right]\left(\eta, \eta'\right) = ()\left[\delta'\right]$ with $\delta \preccurlyeq \delta'$, $\delta(\eta) = \Lambda^T(\eta')$ and $\texttt{dom}(\delta') = \texttt{dom}(\delta) \sqcup \{\eta\}$. In algorithm 1, $\delta(\eta) \leftarrow \Lambda^T(\eta')$ is a shortcut for $\texttt{add\_new\_link}\left[\delta\right]\left(\eta, \eta'\right)$. Initially, to call function `cyclic_unify`, we suppose that δ is well-founded.

Algorithm 1: Algorithm for syntactic unification that can introduce cycles.

```
let rec cyclic_unify η1 η2 =
    (u, v) ← (find^δ_G(η1), find^δ_G(η2));
    if u != v then
        if λ_G(u) ∈ X then
            δ(u) ← Λ^T(v)
        else if λ_G(v) ∈ X then
            δ(v) ← Λ^T(u)
        else if λ_G(u) != λ_G(v) then
            failure
        else In this case, we have: λ_G(u) == λ_G(v) = f/n ∈ F
            δ(u) ← Λ^T(v) ;
            List.iter2 cyclic_unify (γ^i_G(u))_{i=1}^n (γ^i_G(v))_{i=1}^n
```

We observe that, for any linking function δ, the quantity $N_G(\delta) = \texttt{Card}(G \backslash \texttt{dom}(\delta))$ strictly decreases with each call of `cyclic_unify`, which thus ensures its termination. Indeed, each time a new a link is added to some node η, it is a node without link, *i.e.* such that $\eta \notin \texttt{dom}(\delta)$. This is what we are assured by the definition of the quantity $\texttt{find}^\delta_G(\eta)$.

The second observation that can be made is the addition of a link in the case of two nodes with the same function symbol as a label. In reality, this link is only there for complexity reasons and has no influence on the substitution induced by the corresponding linking function. Indeed, in the case where the algorithm `cyclic_unify` succeeds and it has not introduced any δ-cycle, then the terms corresponding to these two nodes are in particular unifiable.

Finally, we also observe that the algorithm is called recursively only in the case of a node without link. As the number of nodes without link is reduced by one for each call, this means that there can be only a linear number of calls to `cyclic_unify`. Actually, due to the $\texttt{find}^\delta_G(\cdot)$ operation, `cyclic_unify` is quadratic in the *worst case* scenario. As explain in [13], this complexity can be improved by modifying functions $\texttt{find}^\delta_G(\cdot)$ and `add_new_link`. In the case of function $\texttt{find}^\delta_G(\cdot)$, it can be done by limiting the depth of the links, *i.e.* by limiting the quantity $M^\delta_G = \max_{\eta \in G} \Gamma^\delta_G(\eta)$. On the other hand, in the case of `add_new_link`, the goal is to

choose the node η that minimises the cardinal of the set $\{\eta' \mid \texttt{find}^{\delta}_{G}(\eta) = \eta\}$. However, this last improvement is costly to maintain, which is why we have chosen to not implement it.

The following theorem expresses the fact that our algorithm is correct in the sense that if the procedure does not introduce a cycle, then we have calculated the *most-general unifier* of the two terms represented by the nodes η_1 and η_2 we give to the algorithm.

Theorem IV.1 (Correctness of `cyclic_unify`) *For all term-graph G, for all nodes $\eta_1, \eta_2 \in G$, for all well-founded linking function δ, if G is δ-cycle free, then*

- *If* $\texttt{cyclic\_unify}\ [\delta]\ (\eta_1, \eta_2)$ *fails then* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable.*
- *Otherwise,* $\texttt{cyclic\_unify}\ [\delta]\ (\eta_1, \eta_2) = ()\,[\delta']$. *In this case, $\delta \preccurlyeq \delta'$ and*
  1. *If G has a δ'-cycle then* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable.*
  2. *Otherwise,*
     (*i*) *There exists a substitution α such that* $\texttt{subst}(\delta') = \texttt{subst}(\delta) \circ \alpha$.
     (*ii*) $\texttt{follow}^{\delta'}_{G}(\eta_1) = \texttt{follow}^{\delta'}_{G}(\eta_2)$, i.e. $\texttt{subst}(\delta')$ *is an unifier of* $\texttt{term}_G(\eta_1)$ *and* $\texttt{term}_G(\eta_2)$, i.e. $\texttt{term}_G(\eta_1)\texttt{subst}(\delta') = \texttt{term}_G(\eta_2)\texttt{subst}(\delta')$.
     (*iii*) α *is the* most-general unifier *of* $\texttt{term}_G(\eta_1)\texttt{subst}(\delta)$ *and* $\texttt{term}_G(\eta_2)\texttt{subst}(\delta)$.

Proof *Proof of this theorem will be given in annex B.1.* □

Now that we have a functional algorithm for unification, it is time to present the algorithm for checking whether a cycle has been introduced or not.

IV.2.2 Step 2: Cycle detection in a term-graph

We can find a δ-cycle in G by a *depth-first search* with function `no_cycle` given in algorithm 2. For this detection algorithm, we want it to be linear in the number of nodes of G in the *worst case* scenario. To do this, we will use the [`HCVisited true`] and [`HCVisited false`] links (modelled respectively by $\Lambda^V(\top)$ and $\Lambda^V(\bot)$). When we start to explore a node, we note it "in progress" (*i.e.* $\Lambda^V(\bot)$) and when the exploration is finished, we note it "finished" (*i.e.* $\Lambda^V(\top)$). Furthermore, if we reach a node with the status "in progress", this means that we have found a δ-cycle. Otherwise, if we reach a node with the status "finished", it means that this node has already been explored and does not produce any δ-cycle.

Algorithm 2: Algorithm for δ-cycle detection in a term-graph.

```
let rec no_cycle η =
    if δ(η) = Λ^V(⊥) then
        failure
    else if δ(η) = Λ^V(⊤) then
        ()
    else if δ(η) = Λ^T(η') then
        δ(η) ← Λ^V(⊥) ;
        no_cycle η' ;
        δ(η) ← Λ^V(⊤)
    else In this case, we have: η ∉ dom(δ)
        if λ(η) = f/n ∈ F then
            δ(η) ← Λ^V(⊥) ;
            List.iter no_cycle (γ^i(η))_{i=1}^n ;
            δ(η) ← Λ^V(⊤)
```

We suppose we have two functions $\texttt{mark\_node}_\bot$ and $\texttt{mark\_node}_\top$ such that if δ is a linking function, η is a node of G and $b \in \{\top, \bot\}$ then $\texttt{mark\_node}_b\,[\delta]\,(\eta) = ()[\delta']$ with $\delta \preccurlyeq \delta'$, $\delta'(\eta) = \Lambda^V(\top)$ and $\texttt{dom}(\delta') = \texttt{dom}(\delta) \cup \{\eta\}$. In algorithm 2, $\delta(\eta) \leftarrow \Lambda^V(b)$ is a shortcut for $\texttt{mark\_node}_b\,[\delta]\,(\eta)$.

For this algorithm, we observe that the quantity

$$\mu(\delta) = \texttt{Card}(G) \;-\; \texttt{Card}(\{\eta \mid \delta(\eta) = \Lambda^V(\bot) \text{ or } \Lambda^V(\top)\})$$

stricly decreases with each call to the function. Moreover, we notice that the links to $\Lambda^V(\top)$ and $\Lambda^V(\bot)$ each verify an invariant which is preserved at any stage of `no_cycle`. These are the two following invariants:

- Let $B(\delta) = \{\eta \mid \delta(\eta) = \Lambda^V(\bot)\}$. The predicate $\text{Inv1}(\delta, \eta, \delta', \eta')$ holds when there exists $\eta_1, \ldots, \eta_n \in G$ such that $B(\delta') = \{\eta_1, \ldots, \eta_n\}$, $\eta_1 = \eta$, $\eta_i \to_\delta \eta_{i+1}$ for all $i \in [\![1\,;\,n-1]\!]$ and $\eta_n \to_\delta \eta'$.
- Let $T(\delta) = \{\eta \mid \delta(\eta) = \Lambda^V(\top)\}$. The predicate $\text{Inv2}(\delta, \delta')$ holds when $\delta \preccurlyeq \delta'$ and for all $\eta \in T(\delta')$, for all η' and η'' such that $\eta \to_\delta^* \eta' \to_\delta^+ \eta''$, we have $\eta' \neq \eta''$.

Because these invariants are preserved at any stage of the `no_cycle` algorithm (see annex B.2 for statement of lemma B.7 and its proof), we can therefore prove its correctness.

Theorem IV.2 (Correctness of `no_cycle`) *Let $\eta \in G$ be a node and δ a well-founded linking function on G. We have the following property:*

$$\texttt{no\_cycle}\,[\delta]\,(\eta) = ()[\delta'] \iff \textit{There is no } \delta\textit{-cycle in } G \textit{ } \delta\textit{-reachable from node } \eta.$$

Proof *Proof will be given in annex B.2.* □

IV.2.3 Final step: Combine the previous algorithms!

First of all, let us notice that the `no_cycle` algorithm has added links to $\Lambda^V(\bot)$ and $\Lambda^V(\top)$. However, these links only have a role for the cycle detection algorithm, we want to remove them at the end if we have indeed not introduced a cycle when calling `cyclic_unify` and thus find the correct induced substitution. Hence, suppose we have a linear function `auto_cleanup` which verifies the following property. Let δ be a linking function on a term-graph G. For a function `func : unit -> 'a` such that $\texttt{func}\,[\delta]\,(()) = \texttt{func}\,()[\delta']$ then $\texttt{auto\_cleanup}\,[\delta]\,(\texttt{func}) = \texttt{func}\,()[\delta]$. *i.e.* `auto_cleanup` saves the linking function δ, executes a function `func` (which possibly creates a new linking function δ' with $\delta \preccurlyeq \delta'$) and finally restores starting linking function δ. If function `func` fails with error `err`, `auto_cleanup` catches this error, restores δ and reproduces `err`. Then, using this special function, we define function `unify_syntactic` by:

Algorithm 3: Final algorithm for syntactic unification.

```
let unify_syntactic η1 η2 =
    cyclic_unify η1 η2;
    auto_cleanup ( fun () → no_cycle η1 )
```

This algorithm first calls the `cyclic_unify` function, then the `no_cycle` function and finally the `auto_cleanup` function to clean links introduced by `no_cycle`.

Finally, we obtain the correctness of the syntactic unification algorithm as an immediate consequence of previous functions correctness theorems.

Theorem IV.3 (Correctness of `unify_syntactic`) *For all term-graph G, for all $\eta_1, \eta_2 \in G$, for all $\delta \in \Delta(G)$ well-founded, if G is δ-cycle free, then*

- *If* `unify_syntactic` $\big[\delta\big]\ \big(\eta_1, \eta_2\big)$ *fails then* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable.*
- *Otherwise,* `unify_syntactic` $\big[\delta\big]\ \big(\eta_1, \eta_2\big) = ()\big[\delta'\big]$. *In this case,* $\delta \preccurlyeq \delta'$ *and*

  (*i*) *There exists an* α *such that* $\texttt{subst}(\delta') = \texttt{subst}(\delta) \circ \alpha$.

  (*ii*) $\texttt{follow}^{\delta'}_{G}(\eta_1) = \texttt{follow}^{\delta'}_{G}(\eta_2)$, i.e. $\texttt{subst}(\delta')$ *is an unifier of* $\texttt{term}_G(\eta_1)$ *and* $\texttt{term}_G(\eta_2)$, i.e.
  $$\texttt{term}_G(\eta_1)\texttt{subst}(\delta') = \texttt{term}_G(\eta_2)\texttt{subst}(\delta').$$

  (*iii*) α *is the* most-general unifier *of* $\texttt{term}_G(\eta_1)\texttt{subst}(\delta)$ *and* $\texttt{term}_G(\eta_2)\texttt{subst}(\delta)$.

Proof *Proof of this theorem will be given in annex B.3.* □

IV.2.4 A short example

To conclude this subsection, we will give an example of our unification algorithm adapted to the structure of directed acyclic graphs. Let us take as an example the terms $f(x, g(y, z))$ and $f(g(y, z), g(a, a))$ for which we have given the corresponding term-graph in figure 7. We define by δ_0 the empty linking function, *i.e.* such that $\texttt{dom}(\delta_0) = \varnothing$. Here is the visual result of the following call:

$$\texttt{unify\_syntactic}\ \big[\delta_0\big]\ \big(\eta_1, \eta_2\big) = ()\big[\delta\big].$$

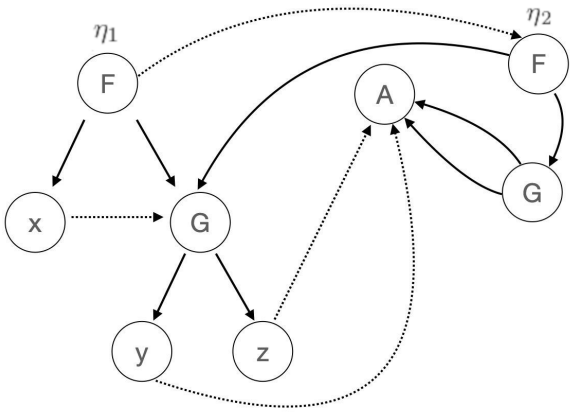

Figure 8: Visual result of an example for `unify_syntactic`

Hence, we have $\texttt{follow}^{\delta}_{G}(\eta_1) = \texttt{follow}^{\delta}_{G}(\eta_2) = f(g(a, a), g(a, a))$. However, terms given by function $\texttt{term}_G(\cdot)$ remains unchanged:

$$\texttt{term}_G(\eta_1) = f(x, g(y, z)) \text{ and } \texttt{term}_G(\eta_2) = f(g(y, z), g(a, a)).$$

Moreover, the substitution induced by δ is $\texttt{subst}(\delta) = \big[x \mapsto g(y, z), y \mapsto a, z \mapsto a\big]$.

IV.3 Formal framework for unification modulo an equational theory

A short presentation of the model of Dolev-Yao was given in the first section: cryptographic primitives are black boxes modelled by a set of equations. As an example, functions for encryption `enc` and decryption `dec` are modelled by

$$\texttt{dec}(\texttt{enc}(x,k),k) = x \quad \texttt{enc}(\texttt{dec}(x,k),k) = x \qquad (\mathcal{E}_{\text{enc/dec}})$$

In fact, in order to determine to verify a query, we need to determine whether a clause R is derivable from another clause R'. We therefore need to know whether these two clauses are unifiable modulo an equational theory defining in particular the behaviour of cryptographic primitives.

Formally, we define a set of equations associated to a set of function symbols $f/n \in \mathcal{F}$ as a set $\mathcal{E} = \left\{u_i = v_i\right\}_{i=1}^{m}$ with $u_i, v_i \in \mathcal{T}(\mathcal{X}, \mathcal{F})$ for each $i \in [\![1\ ;\ n]\!]$. We define equality modulo the equational theory of $\mathcal{E}$ as an equivalence relation over terms in $\mathcal{T}(\mathcal{X}, \mathcal{F})$ such that for all $i \in [\![1\ ;\ m]\!]$, $u_i =_{\mathcal{E}} v_i$. Moreover, this relation is closed by substitution (*i.e.* for any substitution σ, if $t =_{\mathcal{E}} u$ then $t\sigma =_{\mathcal{E}} u\sigma$) and closed by context application (*i.e.* for all $f/n \in \mathcal{F}$, for all terms $t_1, \ldots, t_n, u \in \mathcal{T}(\mathcal{X}, \mathcal{F})$ and for all $i \in [\![1\ ;\ n]\!]$, if $t_i =_{\mathcal{E}} u$ then $f(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n) =_{\mathcal{E}} f(t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_n)$).

Actually, handling an equational theory $\mathcal{E}$ directly in a Horn clause is difficult, this is why ProVerif translate $\mathcal{E}$ into a set of rewriting rules. Hence, to handle those equations during translation of a protocol to a set of Horn clauses, we translate from a signature with equations to a signature without equations. With it, verification can continue to rely on ordinary syntactic unification, and remains very efficient. Formally, each function symbol $f/n \in \mathcal{F}$ is associated to a set of rewriting rules, denoted by $\text{def}_{\mathcal{F}}(f/n)$. For all function symbol $f/n \in \mathcal{F}$, we associate a syntactic function symbol denoted by f^*/n. This syntactic function symbol will never be rewritten. We denote by $\mathcal{T}^*(\mathcal{X}, \mathcal{F})$ the set of fully-syntactic terms, that is to say terms with only syntactic function symbols. This notion of syntactic function symbol already existed in ProVerif but was never formalized before. In fact, giving a formal framework for this type of function symbol allowed us to find a bug in the unification algorithm modulo an equational theory.

A rewriting rule is denoted by $f(u_1, \ldots, u_n) \to r$ where $u_1, \ldots, u_n, r \in \mathcal{T}^*(\mathcal{X}, \mathcal{F})$. Also, we suppose that for all $f/n \in \mathcal{F}$, the rule $f(u_1, \ldots, u_n) \to f^*(u_1, \ldots, u_n)$ is always in $\text{def}_{\mathcal{F}}(f/n)$. In particular, we have:

$$\forall\, f/n \in \mathcal{F}, \text{def}_{\mathcal{F}}(f/n) \neq \varnothing.$$

We define a big-step evaluation on terms as a relation $t \Downarrow u$ such that

1. For all syntactic term $u \in \mathcal{T}^*(\mathcal{X}, \mathcal{F})$, $u \Downarrow u$.
2. For all $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{X}, \mathcal{F})$, if $f(u_1, \ldots, u_n) \to u \in \text{def}_{\mathcal{F}}(f/n)$ and if there exists a substitution σ such that for all $i \in [\![1\ ;\ n]\!]$, $t_i \Downarrow w_i$ and $w_i = u_i\sigma$, then $f(t_1, \ldots, t_n) \Downarrow u\sigma$.
3. For all $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{X}, \mathcal{F})$, if for all $i \in [\![1\ ;\ n]\!]$, $t_i \Downarrow u_i$ then $f^*(t_1, \ldots, t_n) \Downarrow f^*(u_1, \ldots, u_n)$.

Given a term $t \in \mathcal{T}(\mathcal{X}, \mathcal{F})$, there may exists more than one syntactic term $u \in \mathcal{T}^*(\mathcal{X}, \mathcal{F})$ such that $t \Downarrow u$. Indeed, the term t can be evaluated in several different ways, depending on the number of rewriting rules for each function symbol involved in this term. This is why we define the set $\texttt{nf}(t) = \left\{u \in \mathcal{T}^*(\mathcal{X}, \mathcal{F}) \mid t \Downarrow u\right\}$.

We say *a rewriting system $\mathcal{S}$ models a set of equations $\mathcal{E}$* when

1. The equational theory of $\mathcal{E}$ corresponds to the syntactic equality over the set of syntactic terms $\mathcal{T}^*(\mathcal{X},\mathcal{F})$: for all syntactic terms $u_1, u_2 \in \mathcal{T}^*(\mathcal{X},\mathcal{F})$, $u_1 =_\mathcal{E} u_2 \iff u_1 = u_2$.

2. If the rule $t \to u$ is in $\mathcal{S}$ then $t =_\mathcal{E} u$.

3. For each function symbol $f/n \in \mathcal{F}$ and for each rewriting rule $f(u_1,\ldots,u_n) \to u \in \mathcal{S}$ then $f(u_1,\ldots,u_n) =_\mathcal{E} u$.

4. For each function $f/n \in \mathcal{F}$, if $f(t_1,\ldots,t_n) =_\mathcal{E} t$ then there exists a rewriting rule $f(u_1,\ldots,u_n) \to u \in \mathcal{S}$ and a substitution σ such that $t \Downarrow w$ and $w = u\sigma$, $t_i \Downarrow w_i$ and $w_i = u_i\sigma$ for all $i \in [\![1\,;\,n]\!]$.

As an example, with the encryption/decryption scheme, we can define the following rewriting system

$$\begin{array}{ll} \mathtt{enc}(x,k) \to \mathtt{enc}^*(x,k) & \mathtt{dec}(x,k) \to \mathtt{dec}^*(x,k) \\ \mathtt{enc}(\mathtt{dec}^*(x,k),k) \to x & \mathtt{dec}(\mathtt{enc}^*(x,k),k) \to x \end{array} \qquad (\mathcal{S}_{\text{enc/dec}})$$

Thus, the rewriting system $\mathcal{S}_{\text{enc/dec}}$ models the set of equations $\mathcal{E}_{\text{enc/dec}}$.

The following conjecture expresses the fact that our two notions are consistent: to say that two terms are equal modulo an equational theory is to say that these terms share an identical normal form.

Conjecture IV.1 *For all terms $t_1, t_2 \in \mathcal{T}(\mathcal{X},\mathcal{F})$, we have:*

$$t_1 =_{\mathcal{E}_0} t_2 \iff \exists\, u \in \mathcal{T}^*(\mathcal{X},\mathcal{F}), t_1 \Downarrow u \text{ and } t_2 \Downarrow u.$$

Now we can define what a *most-general unifier* is in the sense of unification modulo an equational theory. As the normal forms of the terms are not unique, it may be that these *most-general unifier* are not unique, that is why we speak rather of *partial most-general unifier*.

Definition IV.5 (Partial most-general unifiers) *Let $\mathcal{E}$ be an equational theory on the set $\mathcal{F}$ of function symbols. Let $u, v \in \mathcal{T}(\mathcal{X},\mathcal{F})$ be two terms. We say $\{\sigma_1,\ldots,\sigma_n\}$ is* a complete set of partial most-general unifiers of u and v relative to the equational theory $\mathcal{E}$ *if*

- *for all $i \in [\![1\,;\,n]\!]$, σ_i is an unifier of u and v: $u\sigma_i =_\mathcal{E} v\sigma_i$.*
- *for all substitution α such that $u\alpha =_\mathcal{E} v\alpha$ then there exists a substitution θ and an integer $i \in [\![1\,;\,n]\!]$ such that $\alpha =_\mathcal{E} \sigma_i \circ \theta$.*

IV.4 Algorithms for unification modulo an equational theory

Naively, we can proceed as following:

- Choose two normal forms for terms u and v and try to syntactically unify results terms;
- If it is a success, we found a *partial most-general unifier*. If it fails, we go back to first bullet point;
- If all choices for rewriting rules result in a failure, u and v are not unifiable in sense of unification modulo an equational theory.

In the following, algorithms will be given in a non-deterministic way. We denote by

choose `res` **from** `func` arg1 ... argN

the selection of some result `res` from the call to `func` *arg1* ... *argN* where `func` is a function of arity N and has type $\texttt{func} : \tau_1 \to \cdots \to \tau_N \to \tau$.

Since the algorithm for unification modulo an equational theory creates new nodes, we add to our notation for functions the term-graph in which the node belongs. If we apply a function `func` on n nodes $(\eta_i)_{i=1}^n$ in the original context of a linking function δ in a term-graph G and its result is `res` in the context of the new linking function δ' associated to the term-graph G', we will write:

$$\texttt{func}\,\big[G,\delta\big]\,\big(\eta_1,\ldots,\eta_n\big) = \texttt{res}\big[G',\delta'\big].$$

Also, we suppose we dispose of a function `fresh_copy` such that $\texttt{fresh\_copy}\,\big[\delta\big]\,\big(\eta\big)$ returns a node η', root of a new graph G' isomorphic to the subgraph of G rooted by η but with variables not already linked renamed, follows links given by the linking function δ and preserves δ, *i.e.* we have

$$\texttt{fresh\_copy}\,\big[G,\delta\big]\,\big(\eta\big) = \eta'\big[G',\delta\big].$$

In order to give a first version of modulo unification of an equational theory based on syntactic unification, we start by defining an algorithm to give a normal form of a term.

Algorithm 4: Algorithm to compute a normal form of a term.

```
let rec normal_form η =
    if δ(η) = Λ^T(η') then
        return normal_form η'
    else if λ_G(η) ∈ 𝒳 then
        return η
    else if λ_G(η) = f/n ∈ ℱ then
        if is_syntactic f then
            choose γ_1, ..., γ_n from normal_form γ^1_G(η), ..., normal_form γ^n_G(η);
            η' := build_node f [γ_1, ..., γ_n];
            return η'
        else
            choose a rule (γ_l, γ_r) from def_ℱ(f/n);
            (γ'_ℓ, γ'_r) := fresh_copy (γ_ℓ, γ_r);
            choose γ'_1, ..., γ'_n from normal_form γ^1_G(η), ..., normal_form γ^n_G(η);
            choose (), ..., () from unify_syntactic γ^1(η'_ℓ) γ'_1, ...,
                                   unify_syntactic γ^n(η'_ℓ) γ'_n;
            return η'_r
```

Conjecture IV.2 (Correctness of `normal_form`) *Let η be a node of a term-graph G and δ be a linking function well-founded over G.*

Let η' be a node of a new term-graph G' associated to a new linking function δ' such that ***choose*** η' ***from*** $\texttt{normal\_form}\,\big[G,\delta\big]\,\big(\eta\big)$. *If G and G' are δ'-cycle free then terms given by $t = \texttt{follow}_G^{\delta'}(\eta)$ and $u = \texttt{follow}_{G'}^{\delta'}(\eta')$ verify $t \Downarrow u$.*

Now that we have given an algorithm for calculating a normal form and stated its correction, we can give the algorithm for modular unification.

Algorithm 5: Version 1 of algorithm for unification modulo an equational theory.

```
let unify_modulo η_1 η_2 =
    choose η'_1 from normal_form η_1;
    choose η'_2 from normal_form η_2;
    unify_syntactic η'_1 η'_2
```

Because function `unify_syntactic` is correct (theorem IV.3), correctness of this algorithm is based on correctness of function `normal_form`. Hence, the following conjecture for correctness of `unify_modulo` derives from the corresponding conjecture for `normal_form`.

Conjecture IV.3 (Correctness of `unify_modulo`) *For all term-graph G, for all nodes $\eta_1, \eta_2 \in G$, for all well-founded linking function δ on G, if G has no δ-cycle, then*

- *If* `unify_modulo` $\big[G, \delta\big]\ (\eta_1, \eta_2)$ *fails then there is no* partial most-general unifier *for terms given by* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$.
- *Otherwise,* `unify_modulo` $\big[G, \delta\big]\ (\eta_1, \eta_2) = ()\big[G', \delta'\big]$ *then*
  - (*i*) *There exists a substitution α such that* $\texttt{subst}(\delta') = \texttt{subst}(\delta) \circ \alpha$.
  - (*ii*) *α is a* partial most-general unifier *of terms* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$.
  - (*iii*) *The set* $\big\{\alpha \mid \texttt{subst}(\delta') = \texttt{subst}(\delta) \circ \alpha \wedge \texttt{unify\_modulo}\ \big[G, \delta\big]\ (\eta_1, \eta_2) = ()\big[G', \delta'\big]\big\}$ *is a complete set of* partial most-general unifiers *of* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$.

To get better performance for this algorithm, we can start by using the same principle as for syntactic unification: namely to split the procedure into 2 intermediate steps. The first step is therefore the modular unification which can introduce cycles and the second step is the verification of the absence of cycles. The other aspect that can be improved is to choose the rewriting rules as the unification proceeds rather than calculating a normal form and applying the syntactic unification algorithm. Indeed, if one chooses the rules as one goes along, one can prune branches of the exploration tree of normal forms that are not syntactically unifiable. A version taking into account these remarks of the algorithm for unification modulo an equational theory is then given in annex C.

V Benchmarks

Protocol	**Number of files**	**Version of** ProVerif				**Gain**	
		2.04		2.04_h1			
		Speed	Memory	Speed	Memory	Speed	Memory
Distribution	134	1 m 13 s	4.4 GB	1 m 20 s	4.2 GB	0.9	1
TLS	4	0 h 22 m	5.6 GB	0 h 16 m	0.5 GB	1.4	10.7
Noise Explorer	42	0 h 16 m	5.4 GB	0 h 15 m	2.4 GB	1.1	2.3
Arinc823	18	0 h 10 m	5.6 GB	0 h 12 m	2.1 GB	0.9	2.7
Signal	13	1 h 7 m	21.9 GB	1 h 10 m	3.3 GB	1	6.7
Neuchatel	9	0 h 5 m	0.1 GB	0 h 6 m	0.1 GB	0.8	1.2
TLS + ECH	-	2 h 48 m	162 GB	2 h 59 m	5.6 GB	0.9	29

Table 1: Benchmarks for speed gain and memory consumption.

In fact, we have also implemented an improved version of matching. We say two terms $t, u \in \mathcal{T}^*(\mathcal{X}, \mathcal{F})$ *matches* when there exists a substitution σ such that $t = u\sigma$. This algorithm is useful for subsumption, as can be seen in the following definition:

Definition V.1 (Subsumption) *We say that $H_1 \Rightarrow C_1$* subsumes *$H_2 \Rightarrow C_2$, and we write $(H_1 \Rightarrow C_1) \sqsupseteq (H_2 \Rightarrow C_2)$, if and only if there exists a substitution σ such that $C_2 = C_1\sigma$ and $H_1\sigma \subseteq H_2$ (in sense of multiset inclusion).*

We will therefore carry out benchmarks with the improved algorithms for syntactic unification and for matching. As can be seen from the table 1, the performance results for the "Distribution" files show that some time has been lost. In fact this category, which contains a large number of files, is composed of files that correspond to small protocols like the Needham-Shroeder protocol given in [section I - figure 1]. This slight loss in time can be explained by the time taken by the algorithms to convert a term of type `term` into a term of type `hcterm`.

In addition, the differences in memory gains between the different file groups have a large variation. This variation is simply explained by the greater or lesser use of algorithms that have not yet been improved. Perhaps the most impressive result is that of the file group implementing a lightweight version of TLS, where the gain was tenfold. This result can be explained by the high message redundancy intrinsic to the TLS design. Indeed, in addition to the improvements made to the algorithms, there is also the "entropy" of the set of messages to take into account. The lower this "entropy" is, i.e. the more redundancy there is in the terms involved in Horn clauses, the greater is the compression in the form of DAG.

Conclusion and future work

Let us look again at the diagrams corresponding to the general algorithm used by ProVerif (given in figure 2) and the details of the saturation procedure (given in figure 5).

First, in section II we saw how to abstract a protocol giving in the core language of ProVerif in a *hash consed* version of Horn clauses, namely as *directed acyclic graphs*, which we call *term-graphs*. Second, in subsection IV.2, we gave a version of the syntactic unification algorithm, specifically designed to work on term-graphs, as well as the proof of its correctness. This algorithm, used during clause resolution, allows us to improve the first step of the saturation procedure. And thirdly, in subsection IV.4, we gave a new version of the algorithm for unification modulo an equational theory. This type of unification is used to determine whether a clause is derived from another, *i.e.* within the verification procedure of the security property. However, although one may believe in the accuracy of this new version, we have not given formal proof of it. For this reason, although this algorithm has been implemented in the ProVerif code, we have chosen not to connect it to the rest of the code yet. This is indeed our first future work.

All in all, this leaves the step of simplifying the Horn clauses and their subsumption to be adapted to work with the new representation. In fact, a new version of matching has already been implemented, which has improved a little the subsumption algorithm. Finally, the number of subterms is not yet fully optimised, the problem given in section II being still to be solved. For this, one idea would be to start by estimating its complexity class, intuitively this problem seems to have a high complexity. Based on this estimation, the approach would then be to give an algorithm approximating the exact solution as much as possible. This expensive algorithm will not be called each time a clause is added to a database. It will rather be called at regular intervals of the number of added clauses.

As the issue of memory management is not specific to the ProVerif tool, a possible extension of this work is to integrate these improvements into other tools for formal verification of security properties.

References

[1] David Baelde et al. "An Interactive Prover for Protocol Verification in the Computational Model". In: *42nd IEEE Symposium on Security and Privacy*. 2021. URL: https://hal.archives-ouvertes.fr/hal-03172119/document.

[2] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 483–502. URL: https://bblanche.gitlabpages.inria.fr/publications/BhargavanBlanchetKobeissiSP2017.pdf.

[3] Bruno Blanchet. "Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier López, and Fabio Martinelli. 2013. URL: https://bblanche.gitlabpages.inria.fr/publications/BlanchetFOSAD14.pdf.

[4] Bruno Blanchet. "CryptoVerif: A Computationally-Sound Security Protocol Verifier". In: 2017. URL: https://bblanche.gitlabpages.inria.fr/CryptoVerif/cryptoverif.pdf.

[5] Bruno Blanchet. *ProVerif manual*. URL: http://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf.

[6] Bruno Blanchet. "Security Protocol Verification: Symbolic and Computational Models". In: *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*. Ed. by Pierpaolo Degano and Joshua D. Guttman. Vol. 7215. Lecture Notes in Computer Science. Springer, 2012, pp. 3–29. URL: https://bblanche.gitlabpages.inria.fr/publications/BlanchetHDR.pdf.

[7] Bruno Blanchet. "The Security Protocol Verifier ProVerif and its Horn Clause Resolution Algorithm". In: *9th Workshop on Horn Clauses for Verification and Synthesis*. Ed. by Open Publishing Association. 2022. URL: http://bblanche.gitlabpages.inria.fr/publications/BlanchetHCVS22.html.

[8] Bruno Blanchet. "Using Horn Clauses for Analyzing Security Protocols". In: *Formal Models and Techniques for Analyzing Security Protocols*. Ed. by Véronique Cortier and Steve Kremer. Vol. 5. Cryptology and Information Security Series. IOS Press, 2011, pp. 86–111. URL: https://bblanche.gitlabpages.inria.fr/publications/BlanchetBook09.html.

[9] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More". In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 69–86. URL: https://bblanche.gitlabpages.inria.fr/publications/BlanchetEtAlSP22.pdf.

[10] Vincent Cheval and Lucca Hirshi. In: *Verification of cryptographic protocols*. Ed. by ECJIM 2021. 2021.

[11] Jean-Christophe Filliâtre and Sylvain Conchon. "Type-safe modular hash-consing". In: *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. Ed. by Andrew Kennedy and François Pottier. 2006. URL: https://www.lri.fr/~filliatr/ftp/publis/hash-consing2.pdf.

[12] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach". In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 435–450. URL: https://bblanche.gitlabpages.inria.fr/publications/KobeissiBhargavanBlanchetEuroSP17.pdf.

[13] Temur Kutsia. In: *Introduction to Unification Theory - Speeding up*. 2016. URL: https://www3.risc.jku.at/education/courses/ss2016/unification/slides/02_Syntactic_Unification_Improved_Algorithms.pdf.

Annexes

A ProVerif **syntax**

$M, N ::=$	terms
x	variable ($x \in \mathcal{X}$)
$a[M_1, \ldots, M_n]$	name ($f_a/n_a \in \mathcal{F}$)
$f(M_1, \ldots, M_n)$	function application ($f/n \in \mathcal{F}$)
$ev ::=$	events
$e(M_1, \ldots, M_k)$	($e \in \mathcal{F}$)
$D ::=$	expressions
M	term
$h(D_1, \ldots, D_k)$	function application ($h \in \mathcal{F}$)
`fail`	failure
$P, Q ::=$	processes
0	nil
$\texttt{out}(N, M); P$	output
$\texttt{in}(N, x); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\texttt{new}\ a; P$	restriction
$\texttt{insert}\ tbl(M_1, \ldots, M_n); P$	table insertion
$\texttt{get}\ tbl(x_1, \ldots, x_n)\ \texttt{suchthat}\ D\ \texttt{then}\ P\ \texttt{else}\ Q$	table lookup
$\texttt{let}\ x = D\ \texttt{then}\ P\ \texttt{else}\ Q$	assignment
$\texttt{event}(ev); P$	event

Figure 9: Syntax of the core language of ProVerif

B Proofs

B.1 Proof of theorem IV.1

Lemma B.1 *Let η be a node of a term-graph G and δ a well-founded linking function. Then*

$$\forall\, \eta' \in \Gamma^{\delta}_{G}(\eta), \texttt{find}^{\delta}_{G}(\eta') = \texttt{find}^{\delta}_{G}(\eta).$$

Moreover, if G is δ-cycle free, then

$$\forall\, \eta' \in \Gamma^{\delta}_{G}(\eta), \texttt{follow}^{\delta}_{G}(\eta') = \texttt{follow}^{\delta}_{G}(\eta).$$

Proof *Let $\delta \in \Delta(G)$ be a well-founded linking function on G. We will show the properties by recursion over n the cardinal of $\Gamma^{\delta}_{G}(\eta)$ for all $\eta \in G$.*

Base case: n = 0. *Let $\eta \in G$ be a node such that $\texttt{Card}(\Gamma^{\delta}_{G}(\eta)) = 1$. In this case we must have $\Gamma^{\delta}_{G}(\eta) = \{\eta\}$. Consequently, we have the recursion property true for $n = 1$.*

Inductive step: let $n \in \mathbb{N}^*$. *Let $\eta \in G$ and suppose we have $\texttt{Card}(\Gamma^{\delta}_{G}(\eta)) = n$. Because $n > 1$, we must have $\delta(\eta) = \Lambda^{T}(\eta')$ with $\eta' \in G$. Therefore, $\Gamma^{\delta}_{G}(\eta) = \{\eta\} \cup \Gamma^{\delta}_{G}(\eta')$ with $\texttt{Card}(\Gamma^{\delta}_{G}(\eta')) = n - 1$. By recursion hypothesis, we have: $\forall\, \eta'' \in \Gamma^{\delta}_{G}(\eta'), \texttt{find}^{\delta}_{G}(\eta'') = \texttt{find}^{\delta}_{G}(\eta')$. Moreover, as $\delta(\eta) = \Lambda^{T}(\eta')$ and by definition of $\texttt{find}^{\delta}_{G}(\cdot)$, we have $\texttt{find}^{\delta}_{G}(\eta) = \texttt{find}^{\delta}_{G}(\eta')$. Hence, for all $\eta' \in \Gamma^{\delta}_{G}(\eta), \texttt{find}^{\delta}_{G}(\eta') = \texttt{find}^{\delta}_{G}(\eta)$. Now suppose G without δ-cycle. Then $\texttt{follow}^{\delta}_{G}(\eta)$ is well defined by lemma IV.1. By recursion hypothesis, we have: $\forall\, \eta'' \in \Gamma^{\delta}_{G}(\eta'), \texttt{follow}^{\delta}_{G}(\eta'') = \texttt{follow}^{\delta}_{G}(\eta'')$. By definition of $\texttt{follow}^{\delta}_{G}(\cdot)$, we have $\texttt{follow}^{\delta}_{G}(\eta) = \texttt{follow}^{\delta}_{G}(\texttt{find}^{\delta}_{G}(\eta))$. However, $\texttt{find}^{\delta}_{G}(\eta) = \texttt{find}^{\delta}_{G}(\eta')$. Consequently, $\texttt{follow}^{\delta}_{G}(\eta) = \texttt{follow}^{\delta}_{G}(\texttt{find}^{\delta}_{G}(\eta')) = \texttt{follow}^{\delta}_{G}(\eta')$. Hence, for all $\eta' \in \Gamma^{\delta}_{G}(\eta)$, $\texttt{follow}^{\delta}_{G}(\eta') = \texttt{follow}^{\delta}_{G}(\eta)$.* □

Lemma B.2 *Let δ be a linking function for G. If δ is well-founded then for all nodes $\eta, \eta' \in G$ such that $\delta(\eta) = \Lambda^{T}(\eta')$, we have $\eta \neq \eta'$.*

Lemma B.3 *Let δ be a linking function for G. Let η be a node of G. Let $\eta' \in \Gamma^{\delta}_{G}(\eta)$ be a node linked to η. Let δ' be a second linking function such that $\delta \preccurlyeq \delta'$.*

Then $\texttt{follow}^{\delta'}_{G}(\eta) = \texttt{follow}^{\delta'}_{G}(\eta')$.

Proof *As η' is in the set $\Gamma^{\delta}_{G}(\eta)$ there exists $\eta_1, \ldots, \eta_n$ such that $\eta_1 = \eta$, $\eta_n = \eta'$ and $\delta(\eta_i) = \Lambda^{T}(\eta_{i+1})$ for all $i \in [\![1\ ;\ n-1]\!]$. Moreover, by hypothesis, we have $\delta \preccurlyeq \delta'$. This means we have $\delta'(\eta_i) = \delta(\eta_i) = \Lambda^{T}(\eta_{i+1})$. Consequently, η' is also in the set $\Gamma^{\delta'}_{G}(\eta)$. Finally, by lemma B.1, we can conclude $\texttt{follow}^{\delta'}_{G}(\eta) = \texttt{follow}^{\delta'}_{G}(\eta')$ which completes the proof.* □

Lemma B.4 *Let $\eta, \eta' \in G$ be two nodes corresponding to the same function symbol f/n. Let δ a well-founded linking function such that $\eta, \eta' \notin \texttt{dom}(\delta)$ and G is δ-cycle free. Let δ' be the linking function such that $\texttt{add\_new\_link}\,[\delta]\,(\eta, \eta') = ()\,[\delta']$. Let σ be an unifier of terms $\texttt{follow}^{\delta}_{G}(\eta)$ and $\texttt{follow}^{\delta}_{G}(\eta')$.*

If G is δ'-cycle free then σ is still an unifier of terms $\texttt{follow}^{\delta'}_{G}(\eta)$ and $\texttt{follow}^{\delta'}_{G}(\eta')$.

Lemma B.5 *Let $\delta \in \Delta(G)$ be a linking function for G. If δ is well-founded then $N_G(\delta) > 0$.*

Proof *Let $\eta \in G$ be a node. Because δ is well-founded, $\texttt{find}^{\delta}_{G}(\eta)$ terminates. Thus, there exists a node $\eta' \in \Gamma^{\delta}_{G}(\eta)$ such that $\texttt{find}^{\delta}_{G}(\eta) = \eta'$ and $\eta' \notin \texttt{dom}(\delta)$ or $\delta(\eta') \in \{\Lambda^{V}(b) \mid b \in \{\top, \bot\}\}$. However, because δ is well-founded, no node is link to a value in $\{\Lambda^{V}(\top), \Lambda^{V}(\bot)\}$. Hence, there exists a node $\eta' \notin \texttt{dom}(\delta)$. Thus, $N_G(\delta) \geqslant 1$.* □

Lemma B.6 *Let η be a node of a term-graph G. Let δ be a well-founded linking function such that G is δ-cycle free. Let η' be a node of G such that $\eta \to_\delta^* \eta'$.*

Then the term $s = \texttt{follow}_G^\delta(\eta')$ is a subterm of $t = \texttt{follow}_G^\delta(\eta)$. Moreover, if $\eta \neq \eta'$, then s is a strict subterm of t.

Proof *We will prove this lemma by induction on relation $\eta \to_\delta^* \eta'$. We note $s = \texttt{follow}_G^\delta(\eta')$ and $t = \texttt{follow}_G^\delta(\eta)$.*

Base case: *$\eta = \eta'$. In this case, we have $s = t$* i.e. *s is a subterm of t.*

Inductive step: there exists a node $\eta'' \in G$ such that $\eta \to_\delta \eta'' \to_\delta^* \eta'$. *By induction hypothesis, s is a subterm of term given by $\texttt{follow}_G^\delta(\eta'')$. Now we have to distinguish two cases according to the definition of $\eta \to_\delta \eta''$.*

- *If $\eta \in \texttt{dom}(\delta)$ and $\delta(\eta) = \Lambda^T(\eta'')$. In this case, by lemma B.1, we have $\texttt{follow}_G^\delta(\eta'') = \texttt{follow}_G^\delta(\eta)$. Thus, s is a subterm of t.*
- *Otherwise, if $\eta \notin \texttt{dom}(\delta)$ and $\lambda_G(\eta) = f/n \in \mathcal{F}$. In this case, there exists $i \in [\![1\ ;\ n]\!]$ such that $\eta'' = \gamma_G^i(\eta)$. Moreover, we have $t = f(\texttt{follow}_G^\delta(\gamma_G^1(\eta)), \ldots, \texttt{follow}_G^\delta(\gamma_G^n(\eta)))$. Thus, as s is a subterm of $\texttt{follow}_G^\delta(\eta'')$, s is a subterm of t.* □

Theorem B.1 (Correctness of `cyclic_unify`) *For all term-graph G, for all nodes $\eta_1, \eta_2 \in G$, for all well-founded linking function δ, if G is δ-cycle free, then*

- *If $\texttt{cyclic\_unify}\ [\delta]\ (\eta_1, \eta_2)$ fails then $\texttt{follow}_G^\delta(\eta_1)$ and $\texttt{follow}_G^\delta(\eta_2)$ are not unifiable.*
- $\texttt{cyclic\_unify}\ [\delta]\ (\eta_1, \eta_2) = ()\,[\delta']$ *then $\delta \preccurlyeq \delta'$ and*
  1. *If G has a δ'-cycle then $\texttt{follow}_G^\delta(\eta_1)$ and $\texttt{follow}_G^\delta(\eta_2)$ are not unifiable.*
  2. *Otherwise,*
     (*i*) *There exists an α such that $\texttt{subst}(\delta') = \texttt{subst}(\delta) \circ \alpha$.*
     (*ii*) $\texttt{follow}_G^{\delta'}(\eta_1) = \texttt{follow}_G^{\delta'}(\eta_2)$, i.e. $\texttt{subst}(\delta')$ *is an unifier of* $\texttt{term}_G(\eta_1)$ *and* $\texttt{term}_G(\eta_2)$, i.e.
     $$\texttt{term}_G(\eta_1)\texttt{subst}(\delta') = \texttt{term}_G(\eta_2)\texttt{subst}(\delta').$$
     (*iii*) *α is the* most-general unifier *of* $\texttt{term}_G(\eta_1)\texttt{subst}(\delta)$ *and* $\texttt{term}_G(\eta_2)\texttt{subst}(\delta)$.

Proof *We will proceed by induction over $N_G(\delta)$. Let $u = \texttt{find}_G^\delta(\eta_1)$ and $v = \texttt{find}_G^\delta(\eta_2)$. As δ is well-founded, the nodes u and v exists. Moreover, by lemma B.5, the base case is for $N_G(\delta) = 1$.*

Base case: *$N_G(\delta) = 1$ In this case, we must have u == v because their exists only one node $\eta_\perp \in G$ such that $\eta_\perp \notin \texttt{dom}(\delta)$. Hence, by definition of algorithm 1, we have* $\texttt{cyclic\_unify}\ [\delta]\ (\eta_1, \eta_2) = ()\,[\delta]$. *Moreover, by hypothesis, G is δ-cycle free. With $\alpha = id$, we obtain $\texttt{subst}(\delta) = \texttt{subst}(\delta) \circ \alpha$ and $\texttt{follow}_G^\delta(\eta_1) = \texttt{follow}_G^\delta(\eta_2)$. Finally, for all σ unifier of terms $\texttt{term}_G(\eta_1)\texttt{subst}(\delta)$ and $\texttt{term}_G(\eta_2)\texttt{subst}(\delta)$, we clearly have $\sigma = id \circ \sigma = \alpha \circ \sigma$. Therefore, $\alpha = id$ is the* most-general unifier *of $\texttt{term}_G(\eta_1)\texttt{subst}(\delta)$ and $\texttt{term}_G(\eta_2)\texttt{subst}(\delta)$. Consequently, induction properties are true for $N_G(\delta) = 1$.*

Inductive step: Let *$N_G(\delta) \in \mathbb{N}^*$. If u == v, we have* $\texttt{cyclic\_unify}\ [\delta]\ (\eta_1, \eta_2) = ()\,[\delta]$ *and we conclude as in base case. Let's focus now on the case where u != v. Firstly, if $\lambda_G(u) = f_1/n_1 \in \mathcal{F}$, $\lambda_G(v) = f_2/n_2 \in \mathcal{F}$ and f_1/n_1 != f_2/n_2, then* $\texttt{cyclic\_unify}\ [\delta]\ (\eta_1, \eta_2)$ *fails. Moreover, we have, by definition of $\texttt{follow}_G^\delta(\cdot)$:*

- $\texttt{follow}^{\delta}_{G}(\eta_1) = f_1(\texttt{follow}^{\delta}_{G}(\gamma^1_G(u)), \ldots, \texttt{follow}^{\delta}_{G}(\gamma^{n_1}_G(u)))$ *and*
- $\texttt{follow}^{\delta}_{G}(\eta_2) = f_2(\texttt{follow}^{\delta}_{G}(\gamma^1_G(v)), \ldots, \texttt{follow}^{\delta}_{G}(\gamma^{n_2}_G(v)))$

As f_1 `!=` f_2, $f_1 \neq f_2$, *thus, terms given by* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable. Otherwise, without loss of generality, we can suppose either* $\lambda_G(u) \in \mathcal{X}$ *or* $\lambda_G(u)$ `==` $\lambda_G(v) = f/n \in \mathcal{F}$. *Hence, in those cases, we have* `add_new_link` $\big[\delta\big]\,(u, v) = ()\big[\delta'\big]$ *with* $\delta \preccurlyeq \delta'$ *by hypothesis on* `add_new_link`.

- *Case* $\lambda_G(u) \in \mathcal{X}$*: In this case, we have* `cyclic_unify` $\big[\delta\big]\,(\eta_1, \eta_2) = ()\big[\delta'\big]$. *We must now analyse whether or not a link has been introduced.*

  – *If there is a* δ'*-cycle in* G. *As* G *is* δ*-cycle free* $\texttt{dom}(\delta') = \texttt{dom}(\delta) \sqcup \{u\}$ *by hypothesis on* `add_new_link` *and* u `!=` v, *one has* $u \rightarrow_{\delta'} v \rightarrow^{+}_{\delta} u$. *However, as* $v \notin \texttt{dom}(\delta)$ *and* $\texttt{dom}(\delta') = \texttt{dom}(\delta) \sqcup \{u\}$, *we have* $v \notin \texttt{dom}(\delta')$. *Consequently, we have* $\lambda_G(v) = f/n \in \mathcal{F}$. *As* $\lambda_G(u) \in \mathcal{X}$, *the variable* $\lambda_G(u)$ *occurs in term*

  $$f(\texttt{follow}^{\delta}_{G}(\gamma^1_G(v)), \ldots, \texttt{follow}^{\delta}_{G}(\gamma^n_G(v))).$$

  Finally, we have $\texttt{follow}^{\delta}_{G}(\eta_1) = \lambda_G(u)$ *and*

  $$\texttt{follow}^{\delta}_{G}(\eta_2) = f(\texttt{follow}^{\delta}_{G}(\gamma^1_G(v)), \ldots, \texttt{follow}^{\delta}_{G}(\gamma^n_G(v))).$$

  Thus, terms given by $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable.*

  – *If* G *is* δ'*-cycle free. In this case, by lemma IV.1, for all node* $\eta \in G$, *quantity* $\texttt{follow}^{\delta'}_{G}(\eta)$ *is well-defined. Let* $\alpha = \big[\lambda_G(u) \mapsto \texttt{follow}^{\delta'}_{G}(u)\big]$.

  (*i*) *Because* $\texttt{dom}(\delta') = \texttt{dom}(\delta) \sqcup \{u\}$ *and by* (2), *one has*

  $$\begin{array}{rcl} \texttt{subst}(\delta') & = & \Big[\lambda_G(\eta) \mapsto \texttt{follow}^{\delta'}_{G}(\eta) \mid \eta \in \texttt{dom}(\delta) \wedge \lambda_G(\eta) \in \mathcal{X} \wedge \ldots\Big] \\ & \circ & \big[\lambda_G(u) \mapsto \texttt{follow}^{\delta'}_{G}(u)\big] \end{array}$$

  Let $\eta \in \texttt{dom}(\delta)$ *such that* $\lambda_G(\eta) \in \mathcal{X}$ *and there exists* $\eta' \in G$ *such that* $\delta'(\eta) = \Lambda^T(\eta')$. *We note* $x = \lambda_G(u)$, $t = \texttt{follow}^{\delta'}_{G}(\eta)$ *and* $s = \texttt{follow}^{\delta}_{G}(\eta)$.
  Case 1: if $s = x \in \mathcal{X}$. *In this case,* $\eta = u$ *and, by definition of* α, $t = x\alpha$.
  Case 2: if $s = y \in \mathcal{X}$ *with* $y \neq x$. *In this case, by definition of* δ', *we have* $t = s = s\alpha$. *Case 3: if* $s = f(s_1, \ldots, s_n)$ *with* $t_i = s_i\alpha$ *for all* $i \in [\![1\,;\,n]\!]$. *In this case, we have* $t = f(t_1, \ldots, t_n)$. *Hence* $t = s\alpha$. *Consequently, by induction, we have* $\texttt{follow}^{\delta'}_{G}(\eta) = \texttt{follow}^{\delta}_{G}(\eta)\alpha$. *Thus* $\texttt{subst}(\delta') = \texttt{subst}(\delta) \circ \alpha$.

  (*ii*) *Secondly, we have*

  $$\begin{array}{rcl} \texttt{follow}^{\delta'}_{G}(\eta_1) & = & \texttt{follow}^{\delta'}_{G}(u) \\ & & \big(\textit{because } u \in \Gamma^{\delta}_{G}(\eta_1),\ \delta \preccurlyeq \delta' \textit{ and by lemma } B.3\big) \\ & = & \texttt{follow}^{\delta'}_{G}(v) \\ & & \big(\textit{because } \delta'(u) = \Lambda^T(v)\big) \\ & = & \texttt{follow}^{\delta'}_{G}(\eta_2) \\ & & \big(\textit{because } v \in \Gamma^{\delta}_{G}(\eta_2),\ \delta \preccurlyeq \delta' \textit{ and by lemma } B.3\big) \end{array}$$

  By lemma IV.2, we have $\texttt{term}_G(\eta_1)\texttt{subst}(\delta') = \texttt{term}_G(\eta_2)\texttt{subst}(\delta')$, i.e. $\texttt{subst}(\delta')$ *is an unifier of* $\texttt{term}_G(\eta_1)$ *and* $\texttt{term}_G(\eta_2)$.

(*iii*) *Let* $x = \lambda_G(u)$ *and* $t = \texttt{term}_G(v)\texttt{subst}(\delta)$. *Let* σ *be an unifier of those two terms. Because* σ *is an unifier, we have* $x\sigma = t\sigma$. *Thus, we have* $x\sigma = x\alpha\sigma$. *Moreover, for all variables* $y \in \mathcal{X}$ *such that* $y \neq x$, *we have* $y\alpha = y$ *and so* $y\alpha\sigma = y\sigma$. *Consequently, we have* $\sigma = \alpha \circ \sigma$ *and finally* α *is the* most-general unifier *of* x *and* t.

- *Case* $\lambda_G(u) \texttt{ == } \lambda_G(v) = f/n \in \mathcal{F}$. *In this case, we have to apply* `cyclic_unify` *to all pair of childrens* $\left((\gamma_G^i(u), \gamma_G^i(v))\right)_{i=1}^n$. *As* $N_G(\delta') = N_G(\delta) - 1$, *we can apply induction hypothesis. We note* t *and* s *terms given by* $t = \texttt{follow}_G^\delta(\eta_1)$ *and* $s = \texttt{follow}_G^\delta(\eta_2)$. *Moreover, we note* t_i *for* $i \in [\![1\ ;\ n]\!]$ *the term given by* $t_i = \texttt{follow}_G^\delta(\gamma_G^i(u))$ *and by* s_i *the term given by* $s_i = \texttt{follow}_G^\delta(\gamma_G^i(v))$. *Finally, we note* $\delta_1 = \delta'$. *Let us discriminate our analysis according to whether the call* $\texttt{cyclic\_unify}\left[\delta\right]\left(\eta_1, \eta_2\right)$ *fails or not.*

  - *If* $\texttt{cyclic\_unify}\left[\delta\right]\left(\eta_1, \eta_2\right)$ *fails. Let* $i \in [\![1\ ;\ n]\!]$ *be the minimal integer such that for all* $1 \leqslant j < i$, $\texttt{cyclic\_unify}\left[\delta_j\right]\left(\gamma_G^j(u), \gamma_G^j(v)\right) = ()\left[\delta_{j+1}\right]$ *and* $\texttt{cyclic\_unify}\left[\delta_i\right]\left(\gamma_G^i(u), \gamma_G^i(v)\right)$ *fails. We must now split our analysis according to the existence of an integer* $j \in [\![1\ ;\ i]\!]$ *such that* G *has a* δ_j*-cycle or not.*

    1. *If* $j = 1$ i.e. *if* G *has a* δ_1*-cycle. Then there exists an integer* $i \in [\![1\ ;\ n]\!]$ *such that* $u \to_{\delta_1} v \to_\delta \gamma_G^i(v) \to_\delta^* u$. *By lemma B.6,* t *is a strict subterm of* s. *If we assume by the absurd* t *and* s *are unifiable, there exists a substitution* σ *such that* $t\sigma = u\sigma$. *In particular, terms given by* $t\sigma$ *and* $s\sigma$ *have the same amount of function symbols. Because* t *is a strict subterm of* s, a fortiori, $t\sigma$ *is a strict subterm of* $s\sigma$. *This is absurd. Consequently, terms given by* $\texttt{follow}_G^\delta(\eta_1)$ *and* $\texttt{follow}_G^\delta(\eta_2)$ *are not unifiable.*
    2. *If* $j \in [\![2\ ;\ i]\!]$. *Let us take the minimal integer for which* G *contains a* δ_j*-cycle. In this case, by induction hypothesis, there exists* $\alpha_1, \ldots, \alpha_{j-1}$ *such that for all* $1 \leqslant k < j$, α_k *is the* most-general unifier *of terms given by* $\texttt{follow}_G^{\delta_k}(\gamma_G^k(u))$ *and* $\texttt{follow}_G^{\delta_k}(\gamma_G^k(v))$. *Moreover, because* G *contains a* δ_j*-cycle, terms given by* $\texttt{follow}_G^{\delta_j}(\gamma_G^j(u))$ *and* $\texttt{follow}_G^{\delta_j}(\gamma_G^j(v))$ *are not unifiable.*
       Let us assume by absurdity that the terms t *and* s *are unifiable. Thus, there exists a substitution* σ *such that* $t\sigma = s\sigma$. *In particular, for all* $k \in [\![1\ ;\ j]\!]$, $t_k\sigma = s_k\sigma$. *Hence, by lemma B.4,* σ *is an unifier of terms given by* $\texttt{follow}_G^{\delta_1}(\gamma_G^1(u))$ *and* $\texttt{follow}_G^{\delta_1}(\gamma_G^1(v))$. *But* α_1 *is the* most-general unifier *of these terms, there is therefore a substitution* σ_1 *such that* $\sigma = \alpha_1 \circ \sigma_1$. *Moreover,* α_1 *is such that* $\texttt{subst}(\delta_2) = \texttt{subst}(\delta_1) \circ \alpha_1$. *Hence, as* σ *is an unifier for* $\texttt{follow}_G^{\delta_1}(\gamma_G^2(u))$ *and* $\texttt{follow}_G^{\delta_1}(\gamma_G^2(v))$, a fortiori, σ_1 *is an unifier for* $\texttt{follow}_G^{\delta_2}(\gamma_G^2(u))$ *and* $\texttt{follow}_G^{\delta_2}(\gamma_G^2(v))$. *Doing this for all integers* $k \in [\![1\ ;\ j-1]\!]$, *there exists a substitution* σ_{j-1} *such that* $\sigma = \alpha_1 \circ \cdots \circ \alpha_{j-1} \circ \sigma_{j-1}$ *and* σ_{j-1} *is an unifier for terms given by* $\texttt{follow}_G^{\delta_j}(\gamma_G^j(u))$ *and* $\texttt{follow}_G^{\delta_j}(\gamma_G^j(v))$.
       This is absurd because terms given by $\texttt{follow}_G^{\delta_j}(\gamma_G^j(u))$ *and* $\texttt{follow}_G^{\delta_j}(\gamma_G^j(v))$ *are not unifiable. Consequently, terms given by* $\texttt{follow}_G^\delta(\eta_1)$ *and* $\texttt{follow}_G^\delta(\eta_2)$ *are not unifiable.*
    3. *Finally, if there is no such integer* j. *In this case, by induction hypothesis there exists* $\alpha_1, \ldots, \alpha_{i-1}$ *such that for all* $1 \leqslant j < i$, α_j *is the* most-general unifier *of terms given by* $\texttt{follow}_G^{\delta_j}(\gamma_G^j(u))$ *and* $\texttt{follow}_G^{\delta_j}(\gamma_G^j(v))$. *Moreover, because* $\texttt{cyclic\_unify}\left[\delta_i\right]\left(\gamma_G^i(u), \gamma_G^i(v)\right)$ *fails then terms given by* $\texttt{follow}_G^{\delta_i}(\gamma_G^i(u))$ *and* $\texttt{follow}_G^{\delta_i}(\gamma_G^i(v))$ *are not unifiable. Thus, we conclude this case in the same way as the previous case.*

– *Otherwise,* $\texttt{cyclic\_unify}\left[\delta\right]\left(\eta_1,\eta_2\right)=()\left[\delta_{n+1}\right]$. *In this case, for all* $i\in[\![1\ ;\ n]\!]$, *we have* $\texttt{cyclic\_unify}\left[\delta_i\right]\left(\gamma_G^i(u),\gamma_G^i(v)\right)=()\left[\delta_{i+1}\right]$. *We must now split our analysis according to whether or not* G *has a* δ_{n+1}*-cycle.*

1. *If* G *has a* δ_{n+1}*-cycle. As* G *is* δ*-cycle free and* $\delta\preccurlyeq\delta_1\preccurlyeq\ldots\preccurlyeq\delta_{n+1}$, *there exists an integer* $i\in[\![1\ ;\ n]\!]$ *such that* G *has no* δ_i*-cycle but have a* δ_{i+1}*-cycle. To conclude that the terms* $\texttt{follow}_G^{\delta}(\eta_1)$ *and* $\texttt{follow}_G^{\delta}(\eta_2)$ *are not unifiable, we proceed in the same way as in the previous case when the cycle case is handled.*
2. *If* G *has no* δ_{n+1}*-cycle. Thus,* G *has no* δ_i*-cycle for all* $i\in[\![1\ ;\ n]\!]$. *By induction hypothesis, we have therefore the existence of substitutions* $(\alpha_i)_{i=1}^n$ *such that* $\texttt{subst}(\delta_{i+1})=\texttt{subst}(\delta_i)\circ\alpha_i$ *and* α_i *is the* most-general unifier *of* $\texttt{follow}_G^{\delta_i}(\gamma_G^i(u))$ *and* $\texttt{follow}_G^{\delta_i}(\gamma_G^i(v))$. *Let* $\alpha=\alpha_1\circ\cdots\circ\alpha_n$.

   (*i*) *By definition of* α_i *for all* $i\in[\![1\ ;\ n]\!]$, *we have in particular* $\texttt{subst}(\delta_{i+1})=\texttt{subst}(\delta_i)\circ\alpha_i$. *Thus,* $\texttt{subst}(\delta_{n+1})=\texttt{subst}(\delta_1)\circ\alpha$. *Hence, thanks to lemma B.4, we have* $\texttt{subst}(\delta_{n+1})=\texttt{subst}(\delta)\circ\alpha$.

   (*ii*) *We have*

$$\begin{aligned}\texttt{follow}_G^{\delta_{n+1}}(\eta_1) &= \texttt{follow}_G^{\delta_{n+1}}(u)\\ &\quad\left(\textit{because } u\in\Gamma_G^{\delta}(\eta_1),\delta\preccurlyeq\delta_{n+1}\textit{ and by lemma B.3}\right)\\ &= \texttt{follow}_G^{\delta_{n+1}}(v)\\ &\quad\left(\textit{because }\delta_1(u)=\Lambda^T(v)\textit{ and }\delta_1\preccurlyeq\delta_{n+1}\right)\\ &= \texttt{follow}_G^{\delta_{n+1}}(\eta_2)\\ &\quad\left(\textit{because } v\in\Gamma_G^{\delta}(\eta_2),\delta\preccurlyeq\delta_{n+1}\textit{ and by lemma B.3}\right)\end{aligned}$$

   (*iii*) *Finally, we have to show* α *is the* most-general unifier *of terms* t *and* s. *Let* σ *be an unifier for those terms.*
   By hypothesis, we have in particular $t_1\sigma=s_1\sigma$. *As* α_1 *is the* most-general unifier *of these terms, there is therefore a substitution* σ_1 *such that* $\sigma=\alpha_1\circ\sigma_1$. *Moreover, we have:*

$$\begin{aligned}\texttt{follow}_G^{\delta}(\gamma_G^2(u))\sigma &= \texttt{term}_G(\gamma_G^2(u))\texttt{subst}(\delta)\,(\alpha_1\circ\sigma_1)\\ &\quad(\textit{by lemma IV.2})\\ &= \texttt{term}_G(\gamma_G^2(u))\,(\texttt{subst}(\delta_1)\circ\alpha_1)\,\sigma_1\\ &\quad(\textit{by lemma B.4})\\ &= \texttt{term}_G(\gamma_G^2(u))\texttt{subst}(\delta_2)\sigma_1\\ &\quad(\textit{by induction hypothesis})\\ &= \texttt{follow}_G^{\delta_2}(\gamma_G^2(u))\sigma_1\end{aligned}$$

   Therefore, as $t_2\sigma=s_2\sigma$, σ_1 *is an unifier for terms given by* $\texttt{follow}_G^{\delta_2}(\gamma_G^2(u))$ *and* $\texttt{follow}_G^{\delta_2}(\gamma_G^2(v))$. *Hence, there exists a substitution* σ_2 *such that* $\sigma_1=\alpha_2\circ\sigma_2$. *By iterating this procedure for each integer* $i\in[\![1\ ;\ n]\!]$, *we conclude the existence of a substitution* σ_n *such that* $\sigma=\alpha_1\circ\cdots\circ\alpha_n\circ\sigma_n$.
   Consequently, for all unifier σ *of* t *and* s, *there exists a substitution* σ' *such that* $\sigma=\alpha\circ\sigma'$. *Consequently,* α *is the* most-general unifier *of these two terms.* □

B.2 Proof of lemma IV.2

Lemma B.7 (Preservation of invariants) *Let $\eta_0 \in G$ be a node and δ_0 be a linking function. Let $n \in \mathbb{N}$. Let $P(n)$ be the property given by: for all node $\eta \in G$ and for all linking function δ such that $\mu(\delta) = n$, $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ and $\mathrm{Inv2}(\delta_0, \delta)$ hold then*

- *If* $\texttt{no\_cycle}\left[\delta\right]\left(\eta\right) = ()\left[\delta'\right]$ *then* $\delta'(\eta) = \Lambda^V(\top)$, $\mathrm{Inv1}(\delta_0, \eta_0, \delta', \eta)$ *holds,* $\mathrm{Inv2}(\delta_0, \delta')$ *holds,* $\mu(\delta') \leqslant \mu(\delta)$ *and* $B(\delta) = B(\delta')$.
- *If* $\texttt{no\_cycle}\left[\delta\right]\left(\eta\right)$ *fails then there exists* $\eta' \in G$ *such that* $\eta \to_{\delta_0}^* \eta' \to_{\delta_0}^+ \eta'$.

Proof *Let prove by induction over n the property $P(n)$ for all $n \in \mathbb{N}$.*

Base case: $n = 0$. *Let $\eta \in G$ be a node and δ be a linking function such that $\mu(\delta) = 0$, $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ and $\mathrm{Inv2}(\delta_0, \delta)$ hold. In such a case, we have $\mu(\delta) = 0$ which implies that for all $\eta \in G$, $\delta(\eta) = \Lambda^V(\bot)$ or $\delta(\eta) = \Lambda^V(\top)$. In the former case,* $\texttt{no\_cycle}\left[\delta\right]\left(\eta\right)$ *fails. As $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ holds, there exists $\eta_1, \ldots, \eta_n \in G$ such that $B(\delta) = \{\eta_1, \ldots, \eta_n\}$, $\eta_1 = \eta_0$, $\eta_i \to_{\delta_0} \eta_{i+1}$ for all $i \in [\![1\ ; n-1]\!]$ and $\eta_n \to_{\delta_0} \eta$. Moreover, as $\delta(\eta) = \Lambda^V(\bot)$, we have $\eta \in B(\delta)$,* i.e. *there exists $i_0 \in [\![1\ ; n]\!]$ such that $\eta = \eta_{i_0}$. Then $\eta \to_{\delta_0}^+ \eta$. In the latter case,* $\texttt{no\_cycle}\left[\delta\right]\left(\eta\right) = ()\left[\delta\right]$. *By hypothesis, we have $\delta(\eta) = \Lambda^V(\top)$, $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ holds, $\mathrm{Inv2}(\delta_0, \delta)$ holds, $B(\delta) = B(\delta)$ and $\mu(\delta) \leqslant \mu(\delta)$.*

Inductive step: let $n \in \mathbb{N}^*$. *Let $\eta \in G$ be a node and δ a linking function such that $\mu(\delta) = n$, $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ and $\mathrm{Inv2}(\delta_0, \delta)$ hold.*

- *Case $\delta(\eta) \in \{\Lambda^V(\top), \Lambda^V(\bot)\}$: Similar reasoning as in the base case.*
- *Case $\delta(\eta) = \Lambda^T(\eta')$ for some $\eta' \in G$: In this case, we have* $\texttt{mark\_node}_\bot\left[\delta\right]\left(\eta\right) = ()\left[\delta'\right]$. *Because $\delta(\eta) \notin \{\Lambda^V(\top), \Lambda^V(\bot)\}$ and by property on* $\texttt{mark\_node}_\bot$, *we have $\mu(\delta') = \mu(\delta) - 1 < \mu(\delta)$, $B(\delta') = B(\delta) \sqcup \{\eta\}$ and $T(\delta') = T(\delta)$. By hypothesis on* $\texttt{mark\_node}_\bot$, *we have $\delta \preccurlyeq \delta'$, thus, $\mathrm{Inv2}(\delta_0, \delta')$ holds. Moreover, as $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ holds there exists $\eta_1, \ldots, \eta_n \in G$ such that $B(\delta) = \{\eta_1, \ldots, \eta_n\}$ and $\eta_0 = \eta_1 \to_{\delta_0} \ldots \to_{\delta_0} \eta_n \to_{\delta_0} \eta$. As $\delta(\eta) = \Lambda^T(\eta')$ and $\mathrm{Inv2}(\delta_0, \delta)$ holds then $\eta \to_{\delta_0} \eta'$. As $B(\delta') = \{\eta_1, \ldots, \eta_n, \eta\}$ and $\eta \to_{\delta_0} \eta'$, predicate $\mathrm{Inv1}(\delta_0, \eta_0, \delta', \eta')$ holds. Hence, we can apply induction hypothesis on linking function δ'.*

  - *If* $\texttt{no\_cycle}\left[\delta\right]\left(\eta\right) = ()\left[\delta^{(3)}\right]$. *In this case, we have* $\texttt{no\_cycle}\left[\delta'\right]\left(\eta'\right) = ()\left[\delta''\right]$ *and* $\texttt{mark\_node}_\top\left[\delta''\right]\left(\eta\right) = ()\left[\delta^{(3)}\right]$. *Hence, we have $\delta^{(3)}(\eta) = \Lambda^V(\top)$, $B(\delta^{(3)}) = B(\delta'') \backslash \{\eta\}$ and $T(\delta^{(3)}) = T(\delta'') \cup \{\eta\}$ by hypothesis on* $\texttt{mark\_node}_\top$. *Firstly, we have $\mu(\delta^{(3)}) \leqslant \mu(\delta)$. Indeed*

$$\begin{array}{rcll} \mu(\delta^{(3)}) & = & \mu(\delta'') & (\textit{by hypothesis on } \texttt{mark\_node}_\top) \\ & \leqslant & \mu(\delta') & (\textit{by induction hypothesis}) \\ & < & \mu(\delta) & (\textit{by property on } \delta') \end{array}$$

    Secondly, as $B(\delta') = B(\delta) \sqcup \{\eta\}$, $B(\delta') = B(\delta'')$ by induction hypothesis and $B(\delta^{(3)}) = B(\delta'') \backslash \{\eta\}$, we conclude $B(\delta) = B(\delta^{(3)})$. Thirdly, as $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ holds and $B(\delta) = B(\delta^{(3)})$ then $\mathrm{Inv1}(\delta_0, \eta_0, \delta^{(3)}, \eta)$ holds.
    Finally, we have to show $\mathrm{Inv2}(\delta_0, \delta^{(3)})$ holds. Firstly, as $\mathrm{Inv2}(\delta_0, \delta'')$ holds and by hypothesis on $\texttt{mark\_node}_\top$, *we have $\delta_0 \preccurlyeq \delta^{(3)}$. Because $T(\delta^{(3)}) = T(\delta'') \cup \{\eta\}$ and $\mathrm{Inv2}(\delta_0, \delta'')$ holds, let us focus on the case of $\eta \in T(\delta^{(3)})$. Let $\eta_1, \eta_2 \in G$ be two nodes of G such that $\eta \to_{\delta_0}^* \eta_1 \to_{\delta_0}^+ \eta_2$. There is three possible cases: (a): $\eta_1 = \eta$*

and $\eta_2 = \eta'$, *(b):* $\eta_1 = \eta$ *and* $\eta_1 \to_{\delta_0} \eta' \to^+_{\delta_0} \eta_2$, *(c):* $\eta \to_{\delta_0} \eta' \to^*_{\delta_0} \eta_1 \to^+_{\delta_0} \eta_2$. *In case (a), because* δ_0 *is well-founded and by lemma B.2, we conclude* $\eta \neq \eta'$. *In case (b), suppose briefly that* $\eta_1 = \eta_2$. *This means we have* $\eta' \to^+_{\delta_0} \eta_2 \to_{\delta_0} \eta'$, *which is absurd because* $\mathrm{Inv2}(\delta_0, \delta'')$ *holds and* $\eta' \in T(\delta'')$. *Therefore,* $\eta_1 \neq \eta_2$ *in case (b). Otherwise, in case (c), as* $\mathrm{Inv2}(\delta_0, \delta'')$ *holds by induction hypothesis, we have* $\eta_1 \neq \eta_2$. *Consequently, for all nodes* $\eta_1, \eta_2 \in G$ *such that* $\eta \to^*_{\delta_0} \eta_1 \to^+_{\delta_0} \eta_2$ *then* $\eta_1 \neq \eta_2$. *Thus,* $\mathrm{Inv2}(\delta_0, \delta^{(3)})$ *holds.*

- *Otherwise* `no_cycle` $\big[\delta\big]\ (\eta)$ *fails and so by induction hypothesis, there exists* $\eta'' \in G$ *such that* $\eta' \to^*_{\delta_0} \eta'' \to^+_{\delta_0} \eta''$. *Hence,* $\eta \to_{\delta_0} \eta' \to^*_{\delta_0} \eta'' \to^+_{\delta_0} \eta''$, *wich allows us to conclude.*

- *Case* $\eta \notin \mathtt{dom}(\delta)$ *and* $\lambda_G(\eta) = f/n \in \mathcal{F}$. *In this case, we have* `mark_node`$_\perp \big[\delta\big]\ (\eta) = ()\big[\delta'\big]$. *Hence, by proceeding as in previous case, we can apply induction hypothesis with the linking function* δ'. *Let* $i \in [\![1\ ;\ n-1]\!]$. *Suppose* `no_cycle` $\big[\delta^{(i)}\big]\ \big(\gamma^i_G(\eta)\big) = ()\big[\delta^{(i+1)}\big]$ *and induction hypothesis holds for* $\delta^{(i)}$ *and* $\gamma^i_G(\eta)$. *In particular, we have* $\mathrm{Inv1}(\delta_0, \eta_0, \delta^{(i+1)}, \gamma^i_G(\eta))$ *holds,* $\mathrm{Inv2}(\delta_0, \delta'')$ *holds and* $\mu(\delta^{(i+1)}) \leqslant \mu(\delta^{(i)})$. *Hence, because* $\mu(\delta^{(i+1)}) \leqslant \mu(\delta^{(i)}) \leqslant \ldots \leqslant \mu(\delta') < \mu(\delta)$, *we can apply induction hypothesis for* $\delta^{(i)}$.

  - *If* `no_cycle` $\big[\delta\big]\ (\eta) = ()\big[\delta^{(n+2)}\big]$. *In this case, we have, for all* $i \in [\![1\ ;\ n]\!]$, `no_cycle` $\big[\delta^{(i)}\big]\ \big(\gamma^i_G(\eta)\big) = ()\big[\delta^{(i+1)}\big]$ *and* `mark_node`$_\top \big[\delta^{(n+1)}\big]\ (\eta) = ()\big[\delta^{(n+2)}\big]$. *By induction hypothesis, we have for all* $i \in [\![1\ ;\ n]\!]$, $\delta^{(i)} \preccurlyeq \delta^{(i+1)}$, $\mu(\delta^{(i+1)}) \leqslant \mu(\delta^{(i)})$ *and* $B(\delta^{(i)}) = B(\delta^{(i+1)})$. *Hence,* $\delta' \preccurlyeq \delta^{(n+1)}$, $\mu(\delta^{(n+1)}) \leqslant \mu(\delta')$ *and* $B(\delta^{(n+1)}) = B(\delta')$. *By hypothesis on* `mark_node`$_\top$, *we conclude* $\delta^{(n+2)}(\eta) = \Lambda^V(\top)$, $\delta_0 \preccurlyeq \delta^{(n+2)}$, $\mu(\delta^{(n+2)}) \leqslant \mu(\delta') < \mu(\delta)$ *and* $B(\delta^{(n+2)}) = B(\delta^{(n+1)}) \backslash \{\eta\}$ *and* $B(\delta') = B(\delta) \sqcup \{\eta\}$, *thus,* $B(\delta^{(n+2)}) = B(\delta)$. *Thus,* $\mathrm{Inv1}(\delta_0, \eta_0, \delta, \eta)$ *implies* $\mathrm{Inv1}(\delta_0, \eta_0, \delta^{(n+2)}, \eta)$. *Finally,* $T(\delta^{(n+2)}) = T(\delta^{(n+1)}) \cup \{\eta\}$ *and* $\mathrm{Inv2}(\delta_0, \delta^{(n+1)})$ *holds. Hence, by proceeding as in previous case, we have* $\mathrm{Inv2}(\delta_0, \delta^{(n+2)})$ *holds.*
  - *Otherwise* `no_cycle` $\big[\delta\big]\ (\eta)$ *fails and so there exists an integer* $i_0 \in [\![1\ ;\ n]\!]$ *such that for all* $1 \leqslant j < i_0$, `no_cycle` $\big[\delta^{(j)}\big]\ \big(\gamma^j_G(\eta)\big) = ()\big[\delta^{(j+1)}\big]$ *and* `no_cycle` $\big[\delta^{(i_0)}\big]\ \big(\gamma^{i_0}_G(\eta)\big)$ *fails. By induction hypothesis applied to integer* i_0, *there exists* $\eta_{i_0} \in G$ *such that* $\gamma^{i_0}_G(\eta) \to^*_{\delta_0} \eta_{i_0} \to^+_{\delta_0} \eta_{i_0}$. *Hence,* $\eta \to_{\delta_0} \gamma^{i_0}_G(\eta) \to^*_{\delta_0} \eta_{i_0} \to^+_{\delta_0} \eta_{i_0}$, *which allows us to conclude.*

Consequently, we have prove that property $P(n)$ *holds for all* $n \in \mathbb{N}$. □

Theorem B.2 (Correctness of `no_cycle`) *Let* $\eta \in G$ *be a node and* δ *a well-founded linking function on* G. *We have the following property:*

`no_cycle` $\big[\delta\big]\ (\eta) = ()\big[\delta'\big] \iff$ *There is no* δ*-cycle in* G δ*-reachable from node* η.

Proof *Let* $\eta \in G$ *be a node and* δ *be a well-founded linking function on* G. *Because* δ *is well-founded, we have* $B(\delta) = T(\delta) = \varnothing$. *Moreover, we have* $\delta \preccurlyeq \delta$. *Thus, both* $\mathrm{Inv1}(\eta, \delta, \eta, \delta)$ *and* $\mathrm{Inv2}(\delta, \delta)$ *hold. We can thus apply lemma B.7.*

$\Longrightarrow$ *Suppose* `no_cycle` $\big[\delta\big]\ (\eta) = ()\big[\delta'\big]$. *By property* (P1), *we have* $\delta'(\eta) = \Lambda^V(\top)$. *Thus,* $\eta \in T(\delta')$. *However, as* $\mathrm{Inv2}(\delta, \delta')$ *holds, we have for all nodes* η' *and* η'' *of* G *such that* $\eta \to^*_\delta \eta' \to^+_\delta \eta''$ *then* $\eta' \neq \eta''$. *Thus, there is no* δ*-cycle in* G δ*-reachable from* η.

$\Longleftarrow$ *We will proceed by contraposition. Suppose* `no_cycle` $\big[\delta\big]\ (\eta)$ *fails. By property* (P2), *there exists* $\eta' \in G$ *such that* $\eta \to^*_\delta \eta' \to^+_\delta \eta'$. *Then, by definition IV.4 of* δ*-cycle, there exists a* δ*-cycle in* G δ*-reachable from node* η. □

B.3 Proof of theorem IV.3

Theorem B.3 (Correctness of `unify_syntactic`) *For all term-graph G, for all $\eta_1, \eta_2 \in G$, for all $\delta \in \Delta(G)$ well-founded, if G is δ-cycle free, then*

- *If* `unify_syntactic` $\left[\delta\right]\left(\eta_1, \eta_2\right)$ *fails then* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable.*
- *Otherwise,* `unify_syntactic` $\left[\delta\right]\left(\eta_1, \eta_2\right) = ()\left[\delta'\right]$. *In this case, $\delta \preccurlyeq \delta'$ and*

  (*i*) *There exists an α such that* $\texttt{subst}(\delta') = \texttt{subst}(\delta) \circ \alpha$.

  (*ii*) $\texttt{follow}^{\delta'}_{G}(\eta_1) = \texttt{follow}^{\delta'}_{G}(\eta_2)$, i.e. $\texttt{subst}(\delta')$ *is an unifier of* $\texttt{term}_G(\eta_1)$ *and* $\texttt{term}_G(\eta_2)$, i.e.
  $$\texttt{term}_G(\eta_1)\texttt{subst}(\delta') = \texttt{term}_G(\eta_2)\texttt{subst}(\delta').$$

  (*iii*) α *is the* most-general unifier *of* $\texttt{term}_G(\eta_1)\texttt{subst}(\delta)$ *and* $\texttt{term}_G(\eta_2)\texttt{subst}(\delta)$.

Proof *To prove this theorem, let us do a case analysis on the result of call*

$$\texttt{unify\_syntactic}\left[\delta\right]\left(\eta_1, \eta_2\right).$$

- *Firstly, if* `unify_syntactic` $\left[\delta\right]\left(\eta_1, \eta_2\right)$ *fails, then either* `cyclic_unify` $\left[\delta\right]\left(\eta_1, \eta_2\right)$ *fails or* `no_cycle` $\left[\delta\right]\left(\eta_1\right)$ *fails. If its call to* `cyclic_unify` $\left[\delta\right]\left(\eta_1, \eta_2\right)$ *which fails then, by theorem IV.1, terms given by* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable. Otherwise,* `cyclic_unify` $\left[\delta\right]\left(\eta_1, \eta_2\right) = ()\left[\delta'\right]$ *and it is the call to* `no_cycle` $\left[\delta'\right]\left(\eta_1\right)$ *which fails. By theorem IV.2, this means G has a δ'-cycle. Therefore, by theorem IV.1, terms given by* $\texttt{follow}^{\delta}_{G}(\eta_1)$ *and* $\texttt{follow}^{\delta}_{G}(\eta_2)$ *are not unifiable.*
- *Secondly, if* `unify_syntactic` $\left[\delta\right]\left(\eta_1, \eta_2\right) = ()\left[\delta'\right]$ *then there exists δ'' such that* `cyclic_unify` $\left[\delta\right]\left(\eta_1, \eta_2\right) = ()\left[\delta'\right]$ *and* `no_cycle` $\left[\delta'\right]\left(\eta_1\right) = ()\left[\delta''\right]$. *By definition of function* `auto_cleanup`, *the correct resulting linking function for* `unify_syntactic` *is δ'.*

  By theorem IV.2, the term-graph G has no δ'-cycle δ'-reachable from η_1. Hence, let us show G is δ'-cycle free. As G is suppose to be δ-cycle free, G has no δ'-cycle δ'-reachable from η_1 and $\delta \preccurlyeq \delta'$, it remains to be shown that G has no δ'-cycle δ'-reachable from η_2. Let us proceed by absurd: suppose there exists $\eta_0 \in G$ such that $\eta_2 \rightarrow^*_{\delta'} \eta_0 \rightarrow^+_{\delta'} \eta_0$. Because G is δ-cycle free, there exists $\eta', \eta'' \in G$ such that

  - $\eta_2 \rightarrow^*_{\delta'} \eta' \rightarrow_{\delta'} \eta'' \rightarrow^*_{\delta'} \eta_0 \rightarrow^+_{\delta'} \eta_0$ *or* $\eta_2 \rightarrow^*_{\delta'} \eta_0 \rightarrow^*_{\delta'} \eta' \rightarrow_{\delta'} \eta'' \rightarrow^*_{\delta'} \eta_0$.
  - $\eta' \notin \texttt{dom}(\delta)$ *but* $\delta'(\eta') = \Lambda^T(\eta'')$.

  Nevertheless, η'' is δ'-reachable from η_1. Indeed, if η' is δ'-reachable from η_1 then η'' is still δ'-reachable from η_1. Otherwise, as `cyclic_unify` $\left[\delta\right]\left(\eta_1, \eta_2\right) = ()\left[\delta'\right]$ *and $\eta' \notin \texttt{dom}(\delta)$, we conclude that link between η' and η'' has been created by* `cyclic_unify`. *According to algorithm 1, node η'' is δ-reachable from η_1. Because $\delta \preccurlyeq \delta'$, η'' is still δ'-reachable from η_1. Thus, we have proved there exists a node $\eta'' \in G$ such that $\eta_1 \rightarrow^*_{\delta'} \eta'' \rightarrow^*_{\delta'} \eta_0 \rightarrow^+_{\delta} \eta_0$* i.e. *there exists a δ'-cycle δ'-reachable from η_1. This is absurd because, thanks to theorem IV.2, there is no such δ'-cycle. Finally, by theorem IV.1, we achieve the proof.* □

C A better algorithm for unification modulo an equational theory

Algorithm 6: Optimized version of algorithm for unification modulo an equational theory

```
let unify_modulo η1 η2 =
    cyclic_unify_modulo η1 η2;
    no_cycle_modulo η1
let rec cyclic_unify_modulo η1 η2 =
    (u, v) ← (find^δ_G(η1), find^δ_G(η2));
    if u != v then
        if λ_G(u) ∈ 𝒳 then
            δ(u) ← Λ^T(v)
        else if λ_G(v) ∈ 𝒳 then
            δ(v) ← Λ^T(u)
        else
            (* In this case, we have:  λ_G(u) = f1/n1 ∈ ℱ and λ_G(v) = f2/n2 ∈ ℱ
               *)
            if is_syntactic f1 and is_syntactic f2 then
                if f1 != f2 then
                    failure
                else
                    (* Here, we have:  f1/n1 == f2/n2 = f/n                    *)
                    δ(u) ← Λ^T(v);
                    List.iter2 cyclic_unify_modulo (γ^i_G(u))_{i=1}^n (γ^i_G(v))_{i=1}^n
            else
                choose η1' from choose_rule u;
                choose η2' from choose_rule v;
                cyclic_unify_modulo η1' η2';
                δ(u) ← Λ^T(η1');
                δ(v) ← Λ^T(η2')
```

Algorithm 7: Rest of algorithm 6

```
let choose_rule η =
    (* In this function, we suppose that δ(η) = Λ^⊥ and λ_G(η) = f/n ∈ F.  *)
    if is_syntactic f then
        return η
    else
        choose a rule (γ_l, γ_r) from def_F(f/n);
        (γ'_l, γ'_r) := fresh_copy (γ_l, γ_r);
        choose (), ..., () from cyclic_unify_modulo γ^1(η'_l) γ^1(η), ...,
        cyclic_unify_modulo γ^n(η'_l) γ^n(η);
        return η'_r
let rec no_cycle_modulo η =
    if δ(η) = Λ^V(⊥) then
        failure
    else if δ(η) = Λ^V(⊤) then
        ()
    else if δ(η) = Λ^T(η') then
        δ(η) ← Λ^V(⊥);
        no_cycle_modulo η';
        δ(η) ← Λ^V(⊤)
    else if λ_G(η) = f/n ∈ F then
        (* In this case, η is without link:  η ∉ dom(δ).                    *)
        try
            δ(η) ← Λ^V(⊥);
            List.iter no_cycle_modulo (γ^i(η))_{i=1}^n;
            δ(η) ← Λ^V(⊤)
        with failure →
            (* In this case, function no_cycle_modulo fails for childrens,
               so we have to apply rules in def_F(f/n).                     *)
            if is_syntactic f then
                failure
            else
                choose η' from choose_rule η;
                no_cycle_modulo η'
```

D Source code for some algorithms

D.1 Code for cyclic_unify

```
let rec cyclic_unify hct1 hct2 =
    let hcu1 = F.find hct1
    and hcu2 = F.find hct2 in
    if hcu1 != hcu2 then
    match hcu1.hc_desc, hcu2.hc_desc with
    | HCVar v1, HCVar v2 ->
        if v1.hc_universal || not v2.hc_universal
        then link hcu1 (HCTerm hcu2)
        else link hcu2 (HCTerm hcu1)
    | HCVar _, HCFunApp _ -> link hcu1 (HCTerm hcu2)
    | HCFunApp _, HCVar _ -> link hcu2 (HCTerm hcu1)
    | HCFunApp (f1, args1), HCFunApp (f2, args2) ->
        if f1 != f2 then raise Terms.Unify
        else
        begin
            link hcu1 (HCTerm hcu2);
            List.iter2 cyclic_unify args1 args2
        end
```

D.2 Code for no_cycle

```
let rec no_cycle_noclean hct =
    match hct.hc_link with
    | HCVisited false -> raise Terms.Unify
    | HCVisited true -> ()
    | HCTerm hct' ->
        link hct (HCVisited false);
        no_cycle_noclean hct';
        hct.hc_link <- HCVisited true
    | HCNoLink ->
        begin
        match hct.hc_desc with
            | HCVar _ -> ()
            | HCFunApp (_, args) ->
                link hct (HCVisited false);
                List.iter no_cycle_noclean args;
                hct.hc_link <- HCVisited true
        end
    | _ -> Parsing_helper.internal_error "[hcterms.ml >> no_cycle] Unexpected link."
```

D.3 Code for unify_syntactic

```
let unify hct1 hct2 =
    cyclic_unify hct1 hct2 ;
    auto_cleanup ( fun () -> no_cycle_noclean hct )
```

D.4 Code for match_terms

```
let rec match_terms hct1 hct2 = match hct1.hc_link with
    | HCTerm hcu ->
        if hcu != hct2 then raise Terms.NoMatch
    | HCNoLink ->
        begin match hct1.hc_desc, hct2.hc_desc with
            | HCVar _, _ -> link hct1 (HCTerm hct2)
            | HCFunApp(f1,args1), HCFunApp(f2,args2) ->
                if f1 != f2 then raise Terms.NoMatch;
                List.iter2 match_terms args1 args2;
                link hct1 (HCTerm hct2)
            | _ -> raise Terms.NoMatch
        end
    | _ -> Parsing_helper.internal_error "[hcterms.ml >> match_terms] Unexpected link."
```

E Example of the Needham-Shroeder protocol in ProVerif

E.1 Code for the protocol in ProVerif

```
free c:channel.

type pkey.
type skey.
type rand.

fun aenc(bitstring,pkey,rand):bitstring.
fun pk(skey):pkey.

fun dec(bitstring,skey):bitstring
reduc forall u:bitstring, k:skey, r:rand; dec(aenc(u,pk(k),r),k) = u.

let A(sk_a:skey,pkb:pkey) =
    new na:bitstring;
    new r:rand;
    out(c,aenc((pk(sk_a), na),pkb,r));
    in(c,z:bitstring);
    let (xna:bitstring,xnb:bitstring) = dec(z,sk_a) in
    if xna = na
    then
        new r':rand;
        out(c,aenc(xnb,pkb,r'))
    else 0.

let B(sk_b:skey) =
    in(c,x:bitstring);
    let (xpka:pkey,xna:bitstring) = dec(x,sk_b) in
    new r':rand;
    new nb:bitstring;
```

```
    out(c,aenc((xna,nb),xpka,r'));
    in(c,z:bitstring);
    if dec(z,sk_b) = nb
    then 0
    else 0.

query attacker(new nb).

process
    new sk_a:skey;
    new sk_b:skey;
    out(c,(pk(sk_a),pk(sk_b)));
    (!A(sk_a,pk(sk_b)) | !B(sk_b))
```

E.2 Is the attacker able to deduce n_B?

```
-- Query not attacker(nb[x = v,!1 = v_1]) in process 1.
Translating the process into Horn clauses...
Completing...
Starting query not attacker(nb[x = v,!1 = v_1])
goal reachable: attacker(u) && attacker(r_1) && attacker(v) -> attacker(nb_1)
Abbreviations:
nb_1 = nb[x = aenc((pk(v),u),pk(sk_b[]),r_1),!1 = v_1]

Derivation:
Abbreviations:
nb_1 = nb[x = aenc((pk(k),xna_2),pk(sk_b[]),r_1),!1 = @sid]
r'_2 = r'_1[x = aenc((pk(k),xna_2),pk(sk_b[]),r_1),!1 = @sid]

1. We assume as hypothesis that
attacker(k).

2. We assume as hypothesis that
attacker(r_1).

3. The message (pk(sk_a[]),pk(sk_b[])) may be sent to the attacker at output {3}.
attacker((pk(sk_a[]),pk(sk_b[]))).

4. By 3, the attacker may know (pk(sk_a[]),pk(sk_b[])).
Using the function 2-proj-2-tuple the attacker may obtain pk(sk_b[]).
attacker(pk(sk_b[])).

5. We assume as hypothesis that
attacker(xna_2).

6. By 1, the attacker may know k.
Using the function pk the attacker may obtain pk(k).
attacker(pk(k)).
```

```
7. By 6, the attacker may know pk(k).
By 5, the attacker may know xna_2.
Using the function 2-tuple the attacker may obtain (pk(k),xna_2).
attacker((pk(k),xna_2)).

8. By 7, the attacker may know (pk(k),xna_2).
By 4, the attacker may know pk(sk_b[]).
By 2, the attacker may know r_1.
Using the function aenc the attacker may obtain
    aenc((pk(k),xna_2),pk(sk_b[]),r_1).
attacker(aenc((pk(k),xna_2),pk(sk_b[]),r_1)).

9. The message aenc((pk(k),xna_2),pk(sk_b[]),r_1) that the attacker
    may have by 8 may be received at input {17}.
So the message aenc((xna_2,nb_1),pk(k),r'_2) may be sent to the
    attacker at output {21}.
attacker(aenc((xna_2,nb_1),pk(k),r'_2)).

10. By 9, the attacker may know aenc((xna_2,nb_1),pk(k),r'_2).
By 1, the attacker may know k.
Using the function dec the attacker may obtain (xna_2,nb_1).
attacker((xna_2,nb_1)).

11. By 10, the attacker may know (xna_2,nb_1).
Using the function 2-proj-2-tuple the attacker may obtain nb_1.
attacker(nb_1).

12. By 11, attacker(nb_1).
The goal is reached, represented in the following fact:
attacker(nb_1).


A more detailed output of the traces is available with
set traceDisplay = long.

new sk_a: skey creating sk_a_2 at {1}

new sk_b: skey creating sk_b_2 at {2}

out(c, (~M,~M_1)) with ~M = pk(sk_a_2), ~M_1 = pk(sk_b_2) at {3}

in(c, aenc((pk(a),a_1),~M_1,a_2)) with
    aenc((pk(a),a_1),~M_1,a_2) = aenc((pk(a),a_1),pk(sk_b_2),a_2) at {17} in copy a_3

new r'_1: rand creating r'_2 at {19} in copy a_3

new nb: bitstring creating nb_1 at {20} in copy a_3
```

```
out(c, ~M_2) with ~M_2 = aenc((a_1,nb_1),pk(a),r'_2) at {21} in copy a_3

The attacker has the message 2-proj-2-tuple(dec(~M_2,a)) = nb_1.
A trace has been found.
The previous trace falsifies the query, because the query is
simple and the trace corresponds to the derivation.
RESULT not attacker(nb[x = v,!1 = v_1]) is false.

------------------------------------------------------------
Verification summary:

Query not attacker(nb[x = v,!1 = v_1]) is false.

------------------------------------------------------------
```